# Project 2: Alloy Modelling

## 1 A Alloy Model

```
/*
 * Static model: Signatures
 *
 * The model should contain the following (and potentially other) signatures.
 * If necessary, you have to make some of the signatures abstract and
 * make them extend other signatures.
 */

sig Str {}

sig Location {}
sig PassengerLocation extends Location {}
one sig Unknown in PassengerLocation {}
sig AircraftLocation extends Location {}
one sig InAir in AircraftLocation {}

sig Aircraft extends PassengerLocation{
        seats: some Seat
}

sig Airline {
        name: Str,
        aircraft: set Aircraft,
        flights: set Flight
}

sig Airport extends AircraftLocation{
        code: Str
}

sig Booking {
        id: Str,
        flights: some Flight,
        category: Class,
        passengers: some Passenger
}

enum Class {
        FirstClass,
        Business,
        Economy
}

sig Flight {
        number: Str,
        departureAirport: Airport,
        arrivalAirport: Airport,
        departureTime: Time,
        arrivalTime: Time,
        aircraft: Aircraft,
        passengers: some Passenger
}

sig Passenger {
        bookings: some Booking
}

sig RoundTrip extends Booking { }

abstract sig Seat {}
sig EconomySeat extends Seat {}
sig BusinessSeat extends EconomySeat {}
sig FirstClassSeat extends BusinessSeat {}

sig Time {
        after: lone Time
}

/*
```

```
70  * Static model: Constraints
71  */
72
73  fact airlinesUnique {
74          all disjoint l, l': Airline | l.name != l'.name
75  }
76
77  fact airportsUnique {
78          all disjoint a, a': Airport | a.code != a'.code
79  }
80
81  fact bookingsUnique {
82          all disjoint b, b': Booking | b.id != b'.id
83  }
84
85  fact timeNotSelf {
86          all t: Time | t.after != t
87  }
88
89  fact endTimeMustBeLast {
90          all disjoint t,t': Time | (no t.after) => (t in t'.^after)
91  }
92
93  fact timeLinearlyIncreasing {
94          all disjoint t,t': Time | (t in t'.^after) iff not (t' in t.^after)
95  }
96
97  fact seatsBelong {
98          all s: Seat | one a: Aircraft | s in a.seats
99  }
100
101 fact aircraftBelong {
102         all a: Aircraft | one l: Airline | a in l.aircraft
103 }
104
105 fact operatorMustMatchAircraft {
106         all f: Flight, l: Airline | (f.aircraft in l.aircraft) or (not f in l.flights)
107 }
108
109 fact airportsDiffOverFlight {
110         all f: Flight | f.departureAirport != f.arrivalAirport
111 }
112
113 fact timesIncreaseOverFlight {
114         all f: Flight | isBefore[f.departureTime, f.arrivalTime]
115 }
116
117 fact flightsDoNotOverlap {
118         all b: Booking | all disj f,f': b.flights | isBefore[f.arrivalTime, f'.departureTime] or isBefore[
                f'.arrivalTime, f.departureTime]
119 }
120
121 fact aircraftNotOverlap {
122         all a: Aircraft | all disj f,f': getFlights[a] | isBefore[f.arrivalTime, f'.departureTime] or
                isBefore[f'.arrivalTime, f.departureTime]
123 }
124
125 fact roundtripMatches {
126         all r: RoundTrip | getFirstFlight[r].departureAirport = getLastFlight[r].arrivalAirport
127 }
128
129 fact bookingsMatch {
130         all p: Passenger, b: Booking | (b in p.bookings) iff (p in b.passengers)
131 }
132
133 fact atLeastOneAirline {
134         all f: Flight | some l: Airline | f in l.flights
135 }
136
137 fact passengersMatch {
138         all f: Flight | all p: f.passengers | f in p.bookings.flights
139 }
140
141 fact airportNotInAir {
142         all a: Airport | a != InAir
143 }
```

```
144
145 fact aircraftNotUnknown {
146         all a: Aircraft | a != Unknown
147 }
148
149 // I don't think this is entirely correct.
150 fact appropriateSeats {
151         all f: Flight | all p: f.passengers | one s: f.aircraft.seats, b: p.bookings | f in b.flights and
                isAcceptableSeat[s, b.category]
152 }
153
154 /*
155  * Static model: Predicates
156  */
157
158 // True iff t1 is strictly before t2.
159 pred isBefore[t1, t2: Time] {
160         t2 in t1.^after
161 }
162
163 pred isAcceptableSeat[s: Seat, c: Class]{
164         (s in FirstClassSeat and (c = FirstClass)) or
165         (s in BusinessSeat and   (c = FirstClass or c = Business)) or
166         (s in EconomySeat and    (c = FirstClass or c = Business or c = Economy))
167 }
168
169 /*
170  * Static model: Functions
171  */
172
173 // Returns the departure time of the given flight.
174 fun getDeparture[f: Flight]: Time {
175         f.departureTime
176 }
177
178 // Returns the arrival time of the given flight.
179 fun getArrival[f: Flight]: Time {
180         f.arrivalTime
181 }
182
183 // Returns the airport the given flight departs from.
184 fun getOrigin[f: Flight]: Airport {
185         f.departureAirport
186 }
187
188 // Returns the destination airport of the given flight.
189 fun getDestination[f: Flight]: Airport {
190         f.arrivalAirport
191 }
192
193 // Returns the first flight of the given booking.
194 fun getFirstFlight[b: Booking]: Flight {
195         {f: b.flights | no {f': b.flights | (f != f') and isBefore[f'.departureTime, f.departureTime]}}
196 }
197
198 // Returns the last flight of the given booking.
199 fun getLastFlight[b: Booking]: Flight {
200         {f: b.flights | no {f': b.flights | (f != f') and isBefore[f.departureTime, f'.departureTime]}}
201 }
202
203 // Returns all seats of the given aircraft.
204 fun getSeats[a: Aircraft]: set Seat {
205         a.seats
206 }
207
208 // Returns all flights for which is given aircraft is used.
209 fun getFlights[a: Aircraft]: set Flight {
210         {f: Flight | f.aircraft = a}
211 }
212
213 // Returns all bookings booked by the given passenger.
214 fun getBookings[p: Passenger]: set Booking {
215         p.bookings
216 }
217
218 // Returns all flights contained in the given booking.
```

```
219 fun getFlightsInBooking[b: Booking]: set Flight {
220         b.flights
221 }
222
223 /*
224  * Static model: Tests
225  */
226
227 pred show {
228         #Time = 4
229         #Aircraft = 1
230         #Airline = 1
231         #Booking = 1
232         #RoundTrip = 0
233         #Passenger = 1
234         #Flight = 2
235 }
236 run show for 6
237
238 /*
239  * Dynamic model: Functions
240  */
241
242 // Returns the state which comes after the given state.
243 //fun getNextState[s: State]: State {}
244
245 // Returns the location of the given passenger at the given time.
246 //fun getPassengerLocation[t: Time, p: Passenger]: PassengerLocation {}
247
248 // Returns the location of the given aircraft at the given time.
249 //fun getAircraftLocation[t: Time, ac: Aircraft]: AircraftLocation {}
250
251 // Returns the time whose state the given State represents.
252 //fun getTime[s: State]: Time {}
```

# 2 B Instances

# 3 C Alloy Model

```
1  /*
2   * Static model: Signatures
3   *
4   * The model should contain the following (and potentially other) signatures.
5   * If necessary, you have to make some of the signatures abstract and
6   * make them extend other signatures.
7   */
8
9  sig Str {}
10
11 sig Location {}
12 sig PassengerLocation extends Location {}
13 one sig Unknown in PassengerLocation {}
14 sig AircraftLocation extends Location {}
15 one sig InAir in AircraftLocation {}
16
17 sig Aircraft extends PassengerLocation{
18         seats: some Seat
19 }
20
21 sig Airline {
22         name: Str,
23         aircraft: set Aircraft,
24         flights: set Flight
25 }
26
27 sig Airport extends AircraftLocation{
28         code: Str
29 }
30
31 sig Booking {
32         id: Str,
33         flights: some Flight,
34         category: Class,
```

```
35          passengers: some Passenger
36  }
37
38  enum Class {
39          FirstClass,
40          Business,
41          Economy
42  }
43
44  sig Flight {
45          number: Str,
46          departureAirport: Airport,
47          arrivalAirport: Airport,
48          departureTime: Time,
49          arrivalTime: Time,
50          aircraft: Aircraft,
51          passengers: some Passenger
52  }
53
54  sig Passenger {
55          bookings: some Booking
56  }
57
58  sig RoundTrip extends Booking { }
59
60  abstract sig Seat {}
61  sig EconomySeat extends Seat {}
62  sig BusinessSeat extends EconomySeat {}
63  sig FirstClassSeat extends BusinessSeat {}
64
65  sig Time {
66          after: lone Time
67  }
68
69  /*
70   * Dynamic Model Signatures
71   */
72
73  sig State{
74          passengers: set Passenger,
75          aircrafts: set Aircraft,
76          passenger_locations: Passenger -> one PassengerLocation,
77          aircraft_locations: Aircraft -> one AircraftLocation,
78          time: Time
79  }
80
81
82  /*
83   * Static model: Constraints
84   */
85
86  fact airlinesUnique {
87          all disjoint l, l': Airline | l.name != l'.name
88  }
89
90  fact airportsUnique {
91          all disjoint a, a': Airport | a.code != a'.code
92  }
93
94  fact bookingsUnique {
95          all disjoint b, b': Booking | b.id != b'.id
96  }
97
98  fact timeNotSelf {
99          all t: Time | t.after != t
100 }
101
102 fact endTimeMustBeLast {
103         all disjoint t,t': Time | (no t.after) => (t in t'.^after)
104 }
105
106 fact timeLinearlyIncreasing {
107         all disjoint t,t': Time | (t in t'.^after) iff not (t' in t.^after)
108 }
109
110 fact seatsBelong {
```

```
111         all s: Seat | one a: Aircraft | s in a.seats
112 }
113
114 fact aircraftBelong {
115         all a: Aircraft | one l: Airline | a in l.aircraft
116 }
117
118 fact operatorMustMatchAircraft {
119         all f: Flight, l: Airline | (f.aircraft in l.aircraft) or (not f in l.flights)
120 }
121
122 fact airportsDiffOverFlight {
123         all f: Flight | f.departureAirport != f.arrivalAirport
124 }
125
126 fact timesIncreaseOverFlight {
127         all f: Flight | isBefore[f.departureTime, f.arrivalTime]
128 }
129
130 fact flightsDoNotOverlap {
131         all b: Booking | all disj f,f': b.flights | isBefore[f.arrivalTime, f'.departureTime] or isBefore[
            f'.arrivalTime, f.departureTime]
132 }
133
134 fact aircraftNotOverlap {
135         all a: Aircraft | all disj f,f': getFlights[a] | isBefore[f.arrivalTime, f'.departureTime] or
            isBefore[f'.arrivalTime, f.departureTime]
136 }
137
138 fact roundtripMatches {
139         all r: RoundTrip | getFirstFlight[r].departureAirport = getLastFlight[r].arrivalAirport
140 }
141
142 fact bookingsMatch {
143         all p: Passenger, b: Booking | (b in p.bookings) iff (p in b.passengers)
144 }
145
146 fact atLeastOneAirline {
147         all f: Flight | some l: Airline | f in l.flights
148 }
149
150 fact passengersMatch {
151         all f: Flight | all p: f.passengers | f in p.bookings.flights
152 }
153
154 fact airportNotInAir {
155         all a: Airport | a != InAir
156 }
157
158 fact aircraftNotUnknown {
159         all a: Aircraft | a != Unknown
160 }
161
162
163 fact atNoTimePassengerOnTwoFlights {
164         all disj f1, f2: Flight | all t: Time | #{p: Passenger | p in f1.passengers and isInFlight[f1,p,t]
             and isInFlight[f2,p,t] and p in f2.passengers} = 0
165 }
166
167 // I don't think this is entirely correct.
168 fact appropriateSeats {
169         all f: Flight | all p: f.passengers | one s: f.aircraft.seats, b: p.bookings | f in b.flights and
            isAcceptableSeat[s, b.category]
170 }
171
172 /*
173  * Static model: Predicates
174  */
175
176 // True iff t1 is strictly before t2.
177 pred isBefore[t1, t2: Time] {
178         t2 in t1.^after
179 }
180
181 pred isAcceptableSeat[s: Seat, c: Class]{
182         (s in FirstClassSeat and (c = FirstClass)) or
```

```
183         (s in BusinessSeat and   (c = FirstClass or c = Business)) or
184         (s in EconomySeat and    (c = FirstClass or c = Business or c = Economy))
185 }
186
187
188 pred aircraftOnGround[ac: Aircraft, t: Time] {
189          #{f: Flight | isInAir[f,t] and f.aircraft = ac} = 0
190 }
191
192 pred isInAir[f: Flight, t: Time]{
193         isBefore[getDeparture[f],t] and isBefore[t, getArrival[f]]
194 }
195
196 pred isInFlight[f: Flight, p: Passenger, t: Time]{
197         t in getDeparture[f].*after  and getArrival[f] in t.*after and p in f.passengers
198 }
199
200 /*
201  * Static model: Functions
202  */
203
204 // Returns the departure time of the given flight.
205 fun getDeparture[f: Flight]: Time {
206         f.departureTime
207 }
208
209 // Returns the arrival time of the given flight.
210 fun getArrival[f: Flight]: Time {
211         f.arrivalTime
212 }
213
214 // Returns the airport the given flight departs from.
215 fun getOrigin[f: Flight]: Airport {
216         f.departureAirport
217 }
218
219 // Returns the destination airport of the given flight.
220 fun getDestination[f: Flight]: Airport {
221         f.arrivalAirport
222 }
223
224 // Returns the first flight of the given booking.
225 fun getFirstFlight[b: Booking]: Flight {
226         {f: b.flights | no {f': b.flights | (f != f') and isBefore[f'.departureTime, f.departureTime]}}
227 }
228
229 // Returns the last flight of the given booking.
230 fun getLastFlight[b: Booking]: Flight {
231         {f: b.flights | no {f': b.flights | (f != f') and isBefore[f.departureTime, f'.departureTime]}}
232 }
233
234 // Returns all seats of the given aircraft.
235 fun getSeats[a: Aircraft]: set Seat {
236         a.seats
237 }
238
239 // Returns all flights for which is given aircraft is used.
240 fun getFlights[a: Aircraft]: set Flight {
241         {f: Flight | f.aircraft = a}
242 }
243
244 // Returns all bookings booked by the given passenger.
245 fun getBookings[p: Passenger]: set Booking {
246         p.bookings
247 }
248
249 // Returns all flights contained in the given booking.
250 fun getFlightsInBooking[b: Booking]: set Flight {
251         b.flights
252 }
253
254 /*
255  * Static model: Tests
256  */
257
258 pred show {
```

```
259         #Aircraft = 1
260         #Airline = 1
261         #Booking = 1
262         #RoundTrip = 0
263         #Passenger = 1
264         #Flight = 1
265 }
266
267 run show for 6
268 /*
269  * Dynamic model: Constraints
270  */
271
272 fact exactlyOneStateForEveryTime {
273         all disj s1, s2: State | s1.time != s2.time
274         #State = #Time
275 }
276
277 fact aircraftInAirWhileFlight {
278         all f: Flight, t: Time | isInAir[f,t] => getAircraftLocation[t, f.aircraft] in InAir
279 }
280
281 fact aircraftAtDepartureAirport {
282         all f: Flight | getAircraftLocation[getDeparture[f],f.aircraft] = getOrigin[f]
283 }
284
285 fact aircraftAtDestinationAirport {
286         all f: Flight | getAircraftLocation[getArrival[f],f.aircraft] = getDestination[f]
287 }
288
289 fact aircraftLocationOnGroundDoesntChange {
290         all ac: Aircraft, t1, t2: Time | (t1.after = t2 and aircraftOnGround[ac,t1] and aircraftOnGround[
            ac,t2] ) => getAircraftLocation[t1,ac] = getAircraftLocation[t2, ac]
291 }
292
293 fact aircraftOnGroundWhileNotInFlight {
294         all ac: Aircraft, t: Time |aircraftOnGround[ac,t] => getAircraftLocation[t,ac] in Airport
295 }
296
297 fact personInAircraftWhileInFlight {
298         all f: Flight, p: Passenger , t: Time |  isInFlight[f,p,t] => getPassengerLocation[t, p] = f.
            aircraft
299 }
300
301 fact personSomewhereUnknownWhileNotInFlight {
302         all p: Passenger, t:Time | (#{f: Flight | isInFlight[f,p,t]}=0)  => getPassengerLocation[t, p] in
            Unknown
303 }
304
305
306 /*
307  * Dynamic Predicates
308  */
309
310 /*
311  * Dynamic model: Functions
312  */
313
314 // Returns the state which comes after the given state.
315 fun getNextState[s: State]: State {
316         {s1: State | s.time.after = s1.time}
317 }
318
319 //Returns the State corresponding to a Time [added]
320 fun getState[t: Time]: State{
321         {s: State | s.time = t}
322 }
323
324 // Returns the location of the given passenger at the given time.
325 fun getPassengerLocation[t: Time, p: Passenger]: PassengerLocation {
326         getState[t].passenger_locations[p]
327 }
328
329 // Returns the location of the given aircraft at the given time.
330 fun getAircraftLocation[t: Time, ac: Aircraft]: AircraftLocation {
331         getState[t].aircraft_locations[ac]
```

```
332 }
333
334 // Returns the time whose state the given State represents.
335 fun getTime[s: State]: Time {
336         s.time
337 }
338
339
340 /*
341  * Dynamic model: Tests
342  */
343 pred dynamic_instance_1 {
344         some p: Passenger | #{f: Flight | p in f.passengers} > 1
345         #Flight = 3
346         #Passenger = 1
347         #RoundTrip = 1
348         #Airport = 2
349 }
350
351 pred dynamic_instance_2 {
352         some ac: Aircraft, p: Passenger| #{t: Time | getPassengerLocation[t,p] in Unknown and
                getAircraftLocation[t,ac] in InAir} > 0
353         #Booking = 1
354         #Flight = 2
355 }
356
357 pred dynamic_instance_3 {
358         all t: Time | all disj p1,p2: Passenger | getPassengerLocation[t,p1]= getPassengerLocation[t,p2]
359         all disj b: Booking,  p1,p2: Passenger |  p1 in b.passengers => not (p2 in b.passengers)
360         all rt: RoundTrip, b: Booking, p: Passenger | ((b != rt) and (p in rt.passengers)) => not p in b.
                passengers
361         #RoundTrip = 1
362         #Booking = 3
363         #Aircraft = 2
364         #Airport = 2
365         #Passenger = 2
366 }
367
368 run dynamic_instance_1 for 6
369 run dynamic_instance_2 for 6
370 run dynamic_instance_3 for 6
```

# 4 D Instances