

```

module Company
2  open Cars
  open Areas
4  open Users
  open CarsUsageFunctions

6  /**
   SIGNATURES
   */
10 one sig Company {
    // Vehicles
12   cars: some Car,
    // parking areas
14   parkingAreas: some ParkingArea,
    // registered users
16   users: set User,
    carsUsageData: set CarsUsageData
18 }
    // If there is a car owned by the company, there is also
    a parking area
20 // to leave the car
    {#cars > 0 implies #ChargingArea > 0}

22 /**
   FACTS
   */
24 fact vehiclesMustBeOwnedByTheCompany {
    ∀ c: Car | one com: Company | c in com.cars
26 }
28 fact parkingAreasMustBelongToCompany {
    ∀ p: ParkingArea | one com: Company | p in com.
30   parkingAreas
    }
32 fact ∀UsersAreInCompanyUserSet {
    ∀ u: User | one com: Company | u in com.users
34 }

36 /**
   ASSERTS

```

```

38 */
assert ∀UsersAreInCompanyUserSet {
40   ∀ u: User | one com: Company | u in com.users
}
42
check ∀UsersAreInCompanyUserSet for 3 but 8 Int
44
46 /**
    PREDICATES/FUNCTIONS
48 */
pred show() {
50
    #Car > 2
52    #ParkingArea > 2
    #User > 0
54    /*
        #GPSPoint = 0
56    */
}
58
run show for 5 but 8 Int
60 // but 0 Fee, 0 FixedFee, 0 TimeFee, 10 Vehicle

```

```

module Users
2  /**
    SIGNATURES
4  */
abstract sig Person {}
6  sig Visitor extends Person {}
sig User extends Person {
8
    // code: one UserCode,
10   // email: one Email
    // Not interested in other parameters
12 }
14 /**
    PREDICATES/FUNCTIONS
16 */

```

```

18 pred show() {
    #User > 0
19 }
20
21 run show for 25
22
23 /*
24
25 sig Email {}
26 sig UserCode extends Code {}
27
28 fact codesAreUnique {
29      $\forall$  u1, u2: User | (u1 != u2) implies
30         (u1.code != u2.code)
31 }
32 fact userCodesAreAssociatedToOneUser
33 {
34      $\forall$  uc: UserCode | one u: User | uc in u.code
35 }
36 fact userMailsAreUnique {
37      $\forall$  u1, u2: User | (u1 != u2) implies
38         (u1.email != u2.email)
39 }
40
41
42 */

```

```

module Cars
2 open util/boolean

3
4 /*
5     SIGNATURES
6 */
7 sig Car {
8     battery: one Battery,
9     seats: Int,
10    usedSeats: Int,
11    damages: set Damage,
12    currentState: one CarState,
    plugged: one Bool

```

```

14 // code: one CarCode,
15 // plug: one Plug,
16 // currentPosition: one GPSPoint
17 }
18 {
19     seats > 0 and seats ≤ 4
20     usedSeats ≥ 0 and usedSeats ≤ seats
21     currentState != none
22 // Not necessary true, a user can f.e. exit from a car
23 // while still using it
24 // currentState = InUse implies usedSeats > 0
25 currentState != InUse implies usedSeats = 0
26 currentState = Reserved implies battery.
27     statusPercentage ≥ 20
28 currentState = InUse implies battery.statusPercentage
29     > 0
30 (battery.statusPercentage < 20 and
31     currentState != InUse and
32     plugged = False) implies
33     currentState = Unavailable
34 }
35
36 sig Battery {
37     statusPercentage: Int
38 }
39 {
40     statusPercentage ≥ 0 statusPercentage ≤ 100
41 }
42
43 // Check difference b/w enum and abstract
44
45 abstract sig Damage {}
46 sig MajorDamage, MinorDamage extends Damage {}
47
48 abstract sig CarState {}
49 sig Available, Unavailable, Reserved, InUse extends
50     CarState {}
51
52 /*
53 enum Damage {

```

```

50     MajorDamage, MinorDamage
51 }
52
53 enum CarState {
54     Available, Unavailable, Reserved, InUse, Plugged
55 }
56 */
57
58 /*
59     FACTS
60 */
61 fact batteriesMustBeAssociatedToOneVehicle {
62      $\forall$  b: Battery | one c: Car | b in c.battery
63 }
64
65 fact damagesMustBeAssociatedToACar {
66      $\forall$  d: Damage | d in Car.damages
67 }
68
69 fact carStatesMustBeAssociatedToSomeCars {
70     // It reaches the same end goal, but generates an
71     // additional relation
72     // between a state and a car
73     //  $\forall$  cs: CarState | some c: Car | cs in c.currentState
74      $\forall$  cs: CarState | cs in Car.currentState
75 }
76
77 fact majorDamagesImpliesUnavailableCars {
78      $\forall$  c: Car, m: MajorDamage | m in c.damages implies
79         c.currentState = Unavailable
80 }
81
82 /*
83     ASSERTS
84 */
85 assert  $\forall$ MajorDamagedCarsAreUnavailable {
86      $\forall$  m: MajorDamage, c: Car | m in c.damages implies
87         c.currentState = Unavailable
88 }

```

```

90  assert ∀ReservedCarsHasEnoughBattery {
91      ∀ c: Car | c.currentState = Reserved and
92      c.battery.statusPercentage ≥ 20
93  }
94
95  assert noCarInUseHaveZeroBattery {
96      no c: Car | c.currentState = InUse and c.battery.
97      statusPercentage = 0
98  }
99
100  assert ∀
101      CarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable
102      {
103      ∀ c: Car | (c.battery.statusPercentage < 20 and
104      c.currentState != InUse and
105      c.plugged = False) implies
106      c.currentState = Unavailable
107  }
108
109  /*
110  // Not true, a car may be minor damaged but still
111  // available (the
112  // employee has manually set the status to available again
113  )
114  assert ∀CarsUnusedAndMinorDamagedAreUnavailable {
115      ∀ c: Car, m: MinorDamage | m in c.damages implies
116      c.currentState = Unavailable
117  }
118  */
119
120  check ∀MajorDamagedCarsAreUnavailable for 8
121  check ∀ReservedCarsHasEnoughBattery for 8
122  check ∀
123      CarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable
124      for 8
125  check noCarInUseHaveZeroBattery for 8

```

```

122  /*
    PREDICATES/FUNCTIONS
124  */
    pred show() {
126      #Car > 0
      #(Car.currentState & Reserved) = #Car
128      // Car.currentState & Available = none
    /*
130      #InUse > 0
      #Unavailable > 0
132      #Reserved > 0
      #Plugged > 0
134      #Available > 0
      #MajorDamage > 0
136      #MinorDamage > 0
      #Damage > 0
138      1 in Car.usedSeats
      0 in Battery.statusPercentage
140      19 in Battery.statusPercentage
      20 in Battery.statusPercentage
142  */
    }
144
    run show for 3 but 8 Int
146
    /*
148    open Codes
    open GPSUtilities
150
    sig Plug {}
152    sig CarCode extends Code {}

154    fact carCodesAreAssociatedToOneCar {
      ∀ cc: CarCode | one c: Car | cc in c.code
156    }
    fact plugsMustBeAssociatedToOneVehicle {
158      ∀ p: Plug | one c: Car | p in c.plug
    }

160    fact codesAreUniques {

```

```

162   ∀ c1, c2: Car | (c1 != c2) implies (c1.code != c2.code
      )
    }
164 */

```

```

module Areas
2 //open GPSUtilities
open Cars
4
/**
6   SIGNATURES
*/
8 sig Address {}

10 abstract sig CompanyArea {
    address: one Address,
12 // perimeter: one GPSPolygon
}

14
sig ParkingArea extends CompanyArea {
16   parkingCapacity: Int,
    parkedCars: set Car,
18 // May be useful in a while
// metersForNearestChargingStation: Int
20 }
{
22 // metersForNearestChargingStation > 0
    parkingCapacity ≥ 0
24 #parkedCars ≤ parkingCapacity
}

26
sig ChargingArea extends ParkingArea {
28   chargingCapacity: Int,
    chargingCars: set Car
30 }
{
32 // Note that even a charging area stores the distance
    from the
// nearest charging station
34 chargingCapacity > 0

```



```

#chargingCars ≤ chargingCapacity
36 #chargingCars + #parkedCars ≤ parkingCapacity
//A car can't be charging and parked at the same time
38 chargingCars & parkedCars = none
}

40

42 /**
    FACTS
44 */
// This fact also enforce the uniqueness of area address
46 fact areaAddressesAreAssociatedToExaxtlyOneCompanyArea {
    ∀ a: Address | one ar: CompanyArea | a in ar.address
48 }

50 fact
    parkingCapacityZeroCanOnlyBeAssociatedToChargingArea
    {
        ∀ p: ParkingArea | p.parkingCapacity = 0 implies
52     p in ChargingArea
    }

54
// N.B. Implies and not Iff bcz a car in a ParkingArea
    can also be
56 // Unavailable (or Plugged in a ChargingArea
fact
    carStateAvailableOrReservedImpliesCarAtOneParkingArea
    {
58     ∀ c: Car, pa: ParkingArea, ca: ChargingArea |
        (c.currentState = Available or c.currentState =
            Reserved) implies
60     (c in pa.parkedCars or
        c in ca.parkedCars or
62     c in ca.chargingCars)
    }

64

66 fact carStateInUseImpliesCarNotInParkingArea {
    ∀ c: Car, pa: ParkingArea, ca: ChargingArea |
68     c.currentState = InUse implies

```

```

    (c not in pa.parkedCars and c not in ca.chargingCars
    )
70 }

72 // If a car is plugged  $\leq$  it must be in one charging
    area
    fact carStatePluggedIffCarInOneChargingCars {
74      $\forall$  c: Car | one ca: ChargingArea |
        c.plugged = False iff c in ca.chargingCars
76 }

78
    fact carParkedInOneParkingArea {
80      $\forall$  pa1, pa2: ParkingArea |
        (pa1 != pa2 implies
82     pa1.parkedCars & pa2.parkedCars = none) and
        (ParkingArea.parkedCars & ChargingArea.chargingCars)
        = none
84 }

86
    /*
88      $\forall$  c: Car | some pa1, pa2: ParkingArea |
        (c in pa1.chargingCars and
90     c in pa2.chargingCars) implies pa1 = pa2
    */
92

94 /**
    ASSERTS
96 */
    assert sameCarShouldNotBePluggedAtDifferentChargingArea
    {
98      $\forall$  c: Car | one ca: ChargingArea |
        c.plugged = False implies
100     c in ca.chargingCars
    }

102
    check sameCarShouldNotBePluggedAtDifferentChargingArea
        for 5 but 8 Int

```

```

104  /**
106  PREDICATES/FUNCTIONS
108  */
108  pred show() {
109      #Car > 0
110      #ChargingArea > 0
111      #ParkingArea - #ChargingArea > 0
112      #Battery.statusPercentage = #Car
113      #ChargingArea.chargingCars > 0
114      #ParkingArea.parkedCars - #ChargingArea.parkedCars > 0
115      #Damage = 1
116      1 in Car.usedSeats
117      0 in Battery.statusPercentage
118      19 in Battery.statusPercentage
119      20 in Battery.statusPercentage
120  }

122  run show for 3 but 8 Int

124
126  /*
127  Operating areas are places where a user can pick a
128  vehicle.
129  It is composed by  $\forall$  the parking areas and vehicles of
130  this specific zone
131  */
132  /*
133  sig Socket {}
134  sig AreaCode extends Code {}

135
136  sig OperatingArea extends Area {
137      vehicles: set Vehicle,
138      parkingAreas: set ParkingArea
139  }

140  fact areaCodesAreAssociatedToOneArea
141  {
142       $\forall$  ac: AreaCode | one a: Area | ac in a.code
143  }

```

```

142 fact socketMustBeAssociatedToOneChargingArea {
144    $\forall$  s: Socket | one ca: ChargingArea | s in ca.sockets
146 }
  
```

```

module CarUsageFunctions
2 open Cars
open Users
4
  /**
6   SIGNATURES
  */
8 abstract sig CarsUsageData {}

10 sig DrivingData extends CarsUsageData {
    isDriving: User lone -> lone Car,
12    // The minutes passed from the driving start
    ridingMinutes: Int,
14    // The range of minutes in which there is a passenger
    in the car
    passengersMinutesRange: Int
16 }
  {
18    ridingMinutes > 0
    passengersMinutesRange  $\geq$  0
20 }

22 sig ReservationData extends CarsUsageData {
    hasReserved: User lone -> lone Car,
24    // The time passed from the reservation start
    reservationMinutes: Int
26 }
  {
28    reservationMinutes > 0 and reservationMinutes  $\leq$  60
  }
30
32 sig PluggingData {
  }
  
```

```

34 sig EndingRideData {
35   }
36
37   /**
38     FACTS
39   */
40   fact drivenCarsStateShouldBeInUse {
41     ∀ d: DrivingData, c: Car | (d.isDriving).c != none iff
42       c.currentState = InUse
43   }
44
45   fact reservedCarsStateShouldBeReserved {
46     ∀ r: ReservationData, c: Car | (r.hasReserved).c !=
47       none iff
48       c.currentState = Reserved
49   }
50
51   /**
52     ASSERTS
53   */
54   assert ∀DrivenCarsStateIsInUse {
55     ∀ c: Car | one d: DrivingData | c in User.(d.isDriving
56       )
57   }
58
59   assert ∀DrivenCarsHaveADriver {
60     ∀ c: Car, d: DrivingData | c in User.(d.isDriving)
61     implies
62     (d.isDriving).c != none
63   }
64
65   check ∀DrivenCarsStateIsInUse for 5 but 7 Int
66   check ∀DrivenCarsHaveADriver for 5 but 8 int
67
68   /**
69     PREDICATES
70   */

```

```

72  /*
    pred canReserveACar[u: User, c: Car] {
74       $\forall$  r: ReservationData | not u in (r.hasReserved).Car
        and
        (c.currentState = Available or c.currentState =
          Plugged)
76  }

78  pred addReservationData[r, r': ReservationData, u: User,
    c: Car] {
80      (r'.hasReserved = r.hasReserved + u -> c)  $\wedge$ 
        r.reservationMinutes = 0
82  }

84  /*
    pred addDrivingData[d, d': DrivingData, u: User, c: Car]
        {
86
88      (r'.hasReserved = r.hasReserved + u -> c)  $\wedge$ 
        r.reservationMinutes = 0

90  }

92  /*
    fun driveACar[u: User, c: Car]: DrivingData {
94  }

96  fun reserveACar[u: User, c: Car]: ReservationData {
98  }
100 */

102 run canReserveACar for 3 but 8 Int
    run addReservationData for 3 but 8 Int
104
106 pred show() {
    #ReservationData > 0

```

```
108     #DrivingData > 0
      #Car > 2
    }
110
112   run show for 3 but 10 Int
114   /*
115   sig Timestamp{
116     // Fake bcz we only generates a sm∇ number of integers
117     fakeStamp: Int
118   }
119   */
```