```
module Company
open Cars
open Areas
open Users
open CarsUsageFunctions

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

/**
  FACTS
*/
fact vehiclesMustBeOwnedByTheCompany {
  ∀ c: Car | one com: Company | c in com.cars
}
fact parkingAreasMustBelongToCompany {
  ∀ p: ParkingArea | one com: Company |  p in com.
   parkingAreas
}
fact ∀UsersAreInCompanyUserSet {
  ∀ u: User | one com: Company | u in com.users
}

/**
  ASSERTS
```

```
*/
assert ∀UsersAreInCompanyUserSet {
  ∀ u: User | one com: Company | u in com.users
}

check ∀UsersAreInCompanyUserSet for 3 but 8 Int


/**
  PREDICATES/FUNCTIONS
*/
pred show() {

  #Car > 2
  #ParkingArea > 2
  #User > 0
/*
  #GPSPoint = 0
*/
}

run show for 5 but 8 Int
// but 0 Fee, 0 FixedFee, 0 TimeFee, 10 Vehicle
```

```
module Users
/**
  SIGNATURES
*/
abstract sig Person {}
sig Visitor extends Person {}
sig User extends Person {

//  code: one UserCode,
//  email: one Email
  // Not interested in other parameters
}

/**
  PREDICATES/FUNCTIONS
*/
```

```
pred show() {
  #User > 0
}

run show for 25

/*

sig Email {}
sig UserCode extends Code {}

fact codesAreUnique {
  ∀ u1, u2: User | (u1 != u2) implies
    (u1.code != u2.code)
}
fact userCodesAreAssociatedToOneUser
{
  ∀ uc: UserCode | one u: User | uc in u.code
}
fact userMailsAreUnique {
  ∀ u1, u2: User | (u1 != u2) implies
    (u1.email != u2.email)
}


*/
```

```
module Cars
open util/boolean

/*
  SIGNATURES
*/
sig Car {
  battery: one Battery,
  seats: Int,
  usedSeats: Int,
  damages: set Damage,
  currentState: one CarState,
  plugged: one Bool
```

```
//  code: one CarCode,
//  plug: one Plug,
//  currentPosition: one GPSPoint
}
{
  seats > 0 and seats ≤ 4
  usedSeats ≥ 0 and usedSeats ≤ seats
  currentState != none
//  Not necessary true, a user can f.e. exit from a car
   while still using it
//  currentState = InUse implies usedSeats > 0
  currentState != InUse implies usedSeats = 0
  currentState = Reserved implies battery.
   statusPercentage ≥ 20
  currentState = InUse implies battery.statusPercentage
   > 0
  (battery.statusPercentage < 20 and
    currentState != InUse and
    plugged = False) implies
    currentState = Unavailable
}

sig Battery {
  statusPercentage: Int
}
{
  statusPercentage ≥ 0 statusPercentage ≤ 100
}

// Check difference b/w enum and abstract

abstract sig Damage {}
sig MajorDamage, MinorDamage extends Damage {}

abstract sig CarState {}
sig Available, Unavailable, Reserved, InUse extends
   CarState {}

/*
enum Damage {
```

```
  MajorDamage, MinorDamage
}

enum CarState {
  Available, Unavailable, Reserved, InUse, Plugged
}
*/


/*
  FACTS
*/
fact batteriesMustBeAssociatedToOneVehicle {
  ∀ b: Battery | one c: Car | b in c.battery
}

fact damagesMustBeAssociatedToACar {
  ∀ d: Damage | d in Car.damages
}

fact carStatesMustBeAssociatedToSomeCars {
//  It reaches the same end goal, but generates an
   additional relation
//  between a state and a car
//  ∀ cs: CarState | some c: Car | cs in c.currentState
  ∀ cs: CarState | cs in Car.currentState
}

fact majorDamagesImpliesUnavailableCars {
  ∀ c: Car, m: MajorDamage | m in c.damages implies
    c.currentState = Unavailable
}

/*
  ASSERTS
*/
assert ∀MajorDamagedCarsAreUnavailable {
  ∀ m: MajorDamage, c: Car | m in c.damages implies
    c.currentState = Unavailable
}
```

```
90 assert ∀ReservedCarsHasEnoughBattery {
     ∀ c: Car | c.currentState = Reserved and
92     c.battery.statusPercentage ≥ 20
   }
94
   assert noCarInUseHaveZeroBattery {
96   no c: Car | c.currentState = InUse and c.battery.
      statusPercentage = 0
   }
98
   assert ∀
      CarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable
       {
100   ∀ c: Car | (c.battery.statusPercentage < 20 and
       c.currentState != InUse and
102    c.plugged = False) implies
       c.currentState = Unavailable
104 }
106
   /*
108 // Not true, a car may be minor damaged but still
      available (the
   // employee has manu∀y set the status to available again
      )
110 assert ∀CarsUnusedAndMinorDamagedAreUnavailable {
     ∀ c: Car, m: MinorDamage | m in c.damages implies
112    c.currentState = Unavailable
   }
114 */
116
   check ∀MajorDamagedCarsAreUnavailable for 8
118 check ∀ReservedCarsHasEnoughBattery for 8
   check ∀
      CarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable
       for 8
120 check noCarInUseHaveZeroBattery for 8
```

```
/*
  PREDICATES/FUNCTIONS
*/
pred show() {
  #Car > 0
  #(Car.currentState & Reserved) = #Car
//  Car.currentState & Available = none
/*
  #InUse > 0
  #Unavailable > 0
  #Reserved > 0
  #Plugged > 0
  #Available > 0
  #MajorDamage > 0
  #MinorDamage > 0
  #Damage > 0
  1 in Car.usedSeats
  0 in Battery.statusPercentage
  19 in Battery.statusPercentage
  20 in Battery.statusPercentage
*/
}

run show for 3 but 8 Int

/*
open Codes
open GPSUtilities

sig Plug {}
sig CarCode extends Code {}

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

fact codesAreUniques {
```

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

```
module Areas
//open GPSUtilities
open Cars

/**
  SIGNATURES
*/
sig Address {}

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

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

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

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


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

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

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


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

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

// If a car is plugged <=> it must be in one charging
   area
fact carStatePluggedIffCarInOneChargingCars {
  ∀ c: Car | one ca: ChargingArea |
    c.plugged = False iff c in ca.chargingCars
}


fact carParkedInOneParkingArea {
  ∀ pa1, pa2: ParkingArea |
    (pa1 != pa2 implies
    pa1.parkedCars & pa2.parkedCars = none) and
    (ParkingArea.parkedCars & ChargingArea.chargingCars)
    = none
}


/*
  ∀ c: Car | some pa1, pa2: ParkingArea |
    (c in pa1.chargingCars and
    c in pa2.chargingCars) implies pa1 = pa2
*/


/**
  ASSERTS
*/
assert sameCarShouldNotBePluggedAtDifferentChargingArea
   {
  ∀ c: Car | one ca: ChargingArea |
    c.plugged = False implies
    c in ca.chargingCars
}

check sameCarShouldNotBePluggedAtDifferentChargingArea
   for 5 but 8 Int
```

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

122 run show for 3 but 8 Int


124
/*
126   Operating areas are places where a user can pick a
     vehicle.
    It is composed by ∀ the parking areas and vehicles of
     this specific zone
128 */
/*
130 sig Socket {}
sig AreaCode extends Code {}
132
sig OperatingArea extends Area {
134   vehicles: set Vehicle,
    parkingAreas: set ParkingArea
136 }

138 fact areaCodesAreAssociatedToOneArea
{
140   ∀ ac: AreaCode | one a: Area | ac in a.code
}
```

```
142
fact socketMustBeAssociatedToOneChargingArea {
144   ∀ 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 ≥ 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 ≤ 60
}
30
sig PluggingData {
32 }
```

```
34 sig EndingRideData {
}
36
/**
38   FACTS
*/
40 fact drivenCarsStateShouldBeInUse {
   ∀ d: DrivingData, c: Car | (d.isDriving).c != none iff
42     c.currentState = InUse
}
44
fact reservedCarsStateShouldBeReserved {
46   ∀ r: ReservationData, c: Car | (r.hasReserved).c !=
     none iff
       c.currentState = Reserved
48 }

50 /**
   ASSERTS
52 */
assert ∀DrivenCarsStateIsInUse {
54   ∀ c: Car | one d: DrivingData | c in User.(d.isDriving
     )
}
56


58 assert ∀DrivenCarsHaveADriver {
   ∀ c: Car, d: DrivingData | c in User.(d.isDriving)
     implies
60     (d.isDriving).c != none
}
62



64
check ∀DrivenCarsStateIsInUse for 5 but 7 Int
66 check ∀DrivenCarsHaveADriver for 5 but 8 int

68 /**
   PREDICATES
70 */
```

```
/*
pred canReserveACar[u: User, c: Car] {
  ∀ r: ReservationData | not u in (r.hasReserved).Car
   and
  (c.currentState = Available or c.currentState =
   Plugged)
}


pred addReservationData[r, r': ReservationData, u: User,
    c: Car] {
  (r'.hasReserved = r.hasReserved + u -> c) ∧
  r.reservationMinutes = 0
}

/*
pred addDrivingData[d, d': DrivingData, u: User, c: Car]
    {

  (r'.hasReserved = r.hasReserved + u -> c) ∧
  r.reservationMinutes = 0

}

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

}

fun reserveACar[u: User, c: Car]: ReservationData {

}
*/

run canReserveACar for 3 but 8 Int
run addReservationData for 3 but 8 Int

pred show() {
  #ReservationData > 0
```

```
  #DrivingData > 0
  #Car > 2
}

run show for 3 but 10 Int

/*
sig Timestamp{
  // Fake bcz we only generates a sm∀ number of integers
  fakeStamp: Int
}
*/
```