0.1 Alloy

We have used the functionalities provided by the Alloy tool in order to represent the domain assumptions of our System. The model, as we will see, represents a snapshot of the System at a given time. All the interesting part of the code are commented in order to better explain their meaning.

We have also added some interesting predicates to show some possible world which is not in contrast with our assumptions.

0.1.1 Gps Utilities

```
module GeoUtilities

sig GpsPoint {}

sig GpsVolume {
  gpsPoints: some GpsPoint
}

fact differentGpsVolumeShouldDifferForAtLeastOnePoint {
  all disj gv1, gv2: GpsVolume |
    (gv1.gpsPoints + gv2.gpsPoints) -
    (gv1.gpsPoints & gv2.gpsPoints) ≠ none
}

pred show() {
  #GpsVolume > 1
}
run show for 5
```

In this file, we have modelled the GPS positions that our System has to cope with.

Given our domain assumptions, positions are exact for CompanyArea because they are predefined. In the reality, it does mean that each Parking Area or Charging Area has a given and exact set of GPS points denoting the volume it occupies.

On the other hand, GPS positions for Persons and Cars are derived from devices and they are not always accurate. For this reason, we introduced the concept of GpsVolume, consisting of various GpsPoints, and that should be read as "the volume that a Person/Car can occupy at a given moment basing on their GPS coordinates". It basically means that, knowing the GPS coordinates of a person at a given moment, we built a probabilistic assumption of the volume in space he/she is occupying. Obviously the same concept applies for the cars.

For the reasons explained above, in our model we have can have different Persons and/or Cars in the same GpsVolume.

To model the fact that persons or cars are nearby we say that they have to share at least one GpsPoint. So if a Person is inside a Car he/she should have some GpsPoint in common with it; the same concept applies for Cars inside CompanyAreas.

We will clarify these aspects in the following pages.

As a last note, we assume that two different GpsVolumes have at least one different GpsPoint.

A simple world is shown in Figure 1

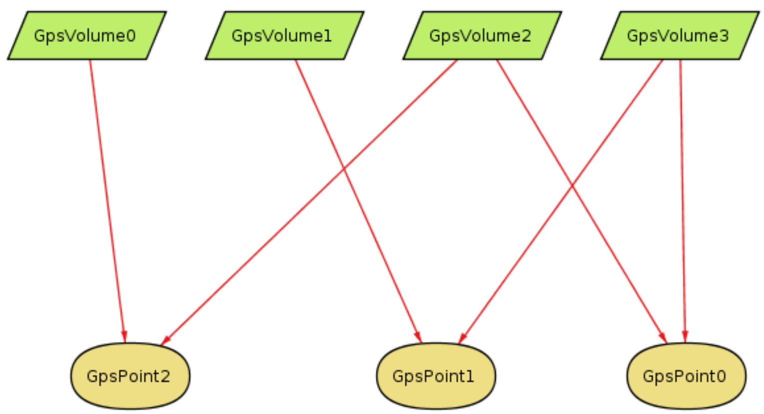

Figure 1: A Gps World

0.1.2 Persons

```
module Persons
open GeoUtilities

/**
  SIGNATURES
*/
```

```
sig Person {
  // We assume that each Person is identified by only one point
  personGpsVolume: one GpsVolume
}
sig User extends Person {}

/**
  PREDICATES/FUNCTIONS
*/
pred show() {
  #Person > 3
}
run show for 6

pred showCouldExistOverlappingPersons() {
  #Person > 1
  #User = 0
  some disj p1, p2: Person |
    p1.personGpsVolume = p2.personGpsVolume
  GpsVolume in Person.personGpsVolume
}
run showCouldExistOverlappingPersons for 2

pred showCouldExistNearbyPersons() {
  #Person > 1
  some disj p1, p2: Person |
    p1.personGpsVolume.gpsPoints & p2.personGpsVolume.gpsPoints ≠ none
}
run showCouldExistNearbyPersons for 4
```

In this file, we have modelled the different kind of people that our System should cope with. In our model, we are not interested in Visitor, so we model simply Persons (general people) and Users (Persons registered to our System).

Figure 2 shows a possible world generated by our Alloy code. We note, for example, that *User0* and *Person1* are both linked to *GpsVolume3*: as we have said before, this is not in contrast with our model.

The *showCouldExistNearbyPersons()* predicate is used to show what we

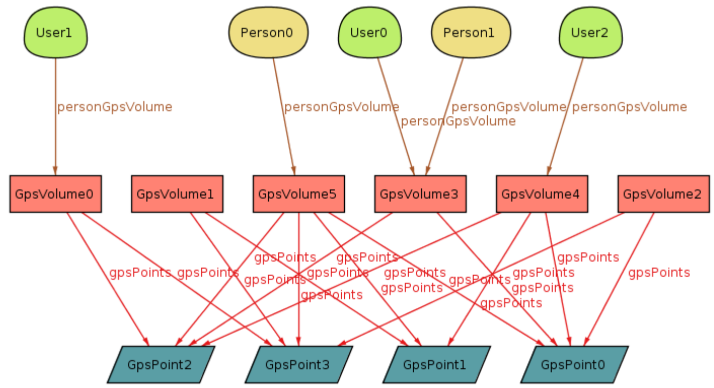

Figure 2: A Persons World

have defined as nearby people: two Persons sharing at least one GpsPoint. This is shown in figure 3, where we can see that *User1* and *User2* are nearby.

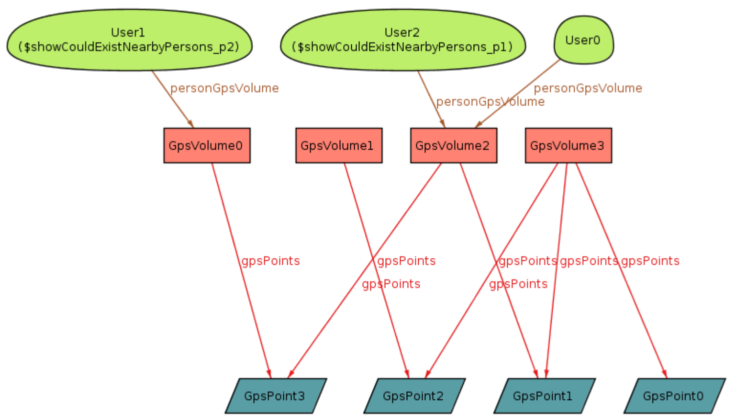

Figure 3: Nearby Persons

0.1.3 Cars

```
module Cars
//open util/boolean
open GeoUtilities
open Persons

/*
```

```
  SIGNATURES
*/
sig Car {
  batteryStatus: one BatteryStatus,
  carSeats: some CarSeat,
  usedSeats: Person lone -> lone carSeats,
  damages: set Damage,
  currentState: one CarState,
  pluggedStatus: one PluggedStatus,
  engineStatus: one EngineStatus,
  carGpsVolume: one GpsVolume
}
{
  (usedSeats.carSeats) ≠ none implies currentState = InUse
  currentState ≠ none
  currentState ≠ InUse implies (usedSeats.carSeats) = none
  currentState = InUse implies pluggedStatus = PluggedOff
  (currentState in Reserved + Available) implies
    batteryStatus = HighBattery
  currentState = InUse implies batteryStatus ≠ ZeroBattery
  (batteryStatus = LowBattery and
    currentState ≠ InUse and
    pluggedStatus = PluggedOff) implies
    currentState = Unavailable
  engineStatus = EngineOn implies currentState = InUse
  currentState ≠ InUse implies engineStatus = EngineOff
}

abstract sig BatteryStatus {}
// Battery less than or greater than 20%
lone sig LowBattery, HighBattery extends BatteryStatus {}
lone sig ZeroBattery extends LowBattery{}

abstract sig EngineStatus {}
lone sig EngineOn, EngineOff extends EngineStatus {}
```

```

abstract sig PluggedStatus {}
lone sig PluggedOn, PluggedOff extends PluggedStatus {}

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

sig CarSeat {}

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

/*
  FACTS
*/
// Trivial relations
fact allEngineStatusAreAssociatedToSomeCar {
  all es: EngineStatus | es in Car.engineStatus
}

fact allPluggedStatusAreAssociatedToSomeCar {
  all ps: PluggedStatus |  ps in Car.pluggedStatus

}

fact allBatteryStatusMustBeAssociatedToSomeCar {
  all b: BatteryStatus | b in Car.batteryStatus
}

fact allCarStatesMustBeAssociatedToSomeCars {
  all cs: CarState | cs in Car.currentState
}

fact allCarSeatsMustBeAssociatedToOneCar {
  all cs: CarSeat | one c: Car | cs in c.carSeats
}

fact damagesMustBeAssociatedToACar {
  all d: Damage | d in Car.damages
```

```
}


// Others
fact personsAreNotUbiquituous {
  all disj c1, c2: Car | no p: Person |
    p in (c1.usedSeats).CarSeat and
    p in (c2.usedSeats).CarSeat
}

fact personsInUsedSeatsHaveSamePositionOfCar {
  all c: Car, p: Person | p in (c.usedSeats).CarSeat implies
    p.personGpsVolume.gpsPoints & c.carGpsVolume.gpsPoints ≠ none
}

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


/**
  ASSERTS
*/
assert allPersonsCantBeInDifferentCars {
  all disj c1, c2: Car | no p: Person |
    p in (c1.usedSeats).CarSeat and p in (c2.usedSeats).CarSeat
}
check allPersonsCantBeInDifferentCars for 10

assert allPersonsInACarMustHaveThatCarPosition {
  all p: Person, c: Car | p in (c.usedSeats).CarSeat implies
    p.personGpsVolume.gpsPoints & c.carGpsVolume.gpsPoints ≠ none
}

```

```
assert allMajorDamagedCarsAreUnavailable {
  all m: MajorDamage, c: Car | m in c.damages implies
    c.currentState = Unavailable
}
check allMajorDamagedCarsAreUnavailable for 10

assert allReservedOrAvailableCarsHaveHighBatteries {
  all c: Car | c.currentState in (Reserved + Available) implies
    c.batteryStatus = HighBattery
}
check allReservedOrAvailableCarsHaveHighBatteries for 3

assert noCarInUseHaveZeroBattery {
  no c: Car | c.currentState = InUse and c.batteryStatus = ZeroBattery
}
check noCarInUseHaveZeroBattery for 10

assert allCarWithUsedSeatsShouldBeInUse {
  all c: Car | (c.usedSeats).CarSeat ≠ none implies c.currentState = InUse
}
check allCarWithUsedSeatsShouldBeInUse for 10

assert allCarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable {
  all c: Car | (c.batteryStatus = LowBattery and
    c.currentState ≠ InUse and
    c.pluggedStatus = PluggedOff) implies
    c.currentState = Unavailable
}
check allCarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable for 10

assert noPluggedCarIsInUse {
  all c: Car | c.currentState = InUse implies c.pluggedStatus = PluggedOff
```

```
}
check noPluggedCarIsInUse for 10

assert allEnginesOnAreAssociatedToInUseCars {
  all c: Car | c.engineStatus = EngineOn implies c.currentState = InUse
}
check allEnginesOnAreAssociatedToInUseCars for 3

assert allUsedSeatsHaveSamePositionOfCars {
  all c: Car | (c.usedSeats).CarSeat ≠ none implies
    (c.usedSeats).(c.carSeats).personGpsVolume.gpsPoints &
     c.carGpsVolume.gpsPoints ≠ none
}
check allUsedSeatsHaveSamePositionOfCars for 3


/*
  PREDICATES/FUNCTIONS
*/
// A car may be perfectly functioning but still unavailable (the external
// employee has manually set the status to Unavailable)
pred showCouldExistSomeUnavailableCarWithNoMajorDamageAndHighBattery {
  #Car > 0
  #Unavailable = #Car
  #MajorDamage = 0
  #LowBattery = 0
  #Person = 0
  GpsVolume in (Car.carGpsVolume + Person.personGpsVolume)

}
run showCouldExistSomeUnavailableCarWithNoMajorDamageAndHighBattery for 3

```

```
pred showCouldExistSomeCarWithLoweBattery {
  #Car > 0
  #LowBattery > 0
}
run showCouldExistSomeCarWithLoweBattery for 3

// A car may have minor damages but still available (the external
// employee has manually set the status to Available)
pred showCouldExistSomeAvailableCarWithMinorDamages {
  #MinorDamage = #Car
  #Available = #Car
}
run showCouldExistSomeAvailableCarWithMinorDamages for 3

// It does mean that a User has turned the engine off outside a parking area
pred showCouldExistSomeInUseCarsWithEngineOff {
  #Car > 0
  #InUse = #Car
  #EngineOff = #Car
}
run showCouldExistSomeInUseCarsWithEngineOff for 3

// Same as before, all the people have left the car, even it is still in use
pred showCouldExistSomeInUseCarsWithEngineOnAndAllPersonsOutside {
  #Car > 0
  #InUse = #Car
  #EngineOn = #Car
  #Person > 0
  #Damage = 0
  #CarSeat = #Car
  #Car.usedSeats = 0
  GpsVolume in (Car.carGpsVolume + Person.personGpsVolume)
}
```

```
run showCouldExistSomeInUseCarsWithEngineOnAndAllPersonsOutside for 6

// Not only users have access to the car. We ensure that a User reserve a Car,
// but we don't know if he/she will use it.
pred showCouldExistSomeInUseCarsWithAllSeatsOccupiedByNonUsers {
  #Car > 0
  #Person > 0
  #User = 0
}
run showCouldExistSomeInUseCarsWithAllSeatsOccupiedByNonUsers for 3

// Show that different people can be in the same car
pred showMorePersonsInOneCar {
  #Car.usedSeats > 1
  #Car = 1
}
run showMorePersonsInOneCar for 7

pred show() {
  #Car > 0
  #Person > 0
  #GpsVolume > 1
  #Car.damages < 3
}
run show for 3
```

In this piece of code we show our model for the Cars managed by our System; a possible world is shown in figure 4. We can note that there is a single car, characterized by

- a PluggedOff status: this is consistent since the car is also InUse;
- an EngingOn status: as for the above, this is consistent since the car is also InUse;

- two different MinorDamages: this is consistent since Users can also use Cars that have minor damages;

- a LowBattery: this is consistent since the car is InUse; when the Car will be parked, its status, according to our assumptions, will be set to Unavailable

- two CarSeats: they are occupied by an User and a Person, that are both nearby our Car (i.e. they have at least one GpsPoint in common with our Car).

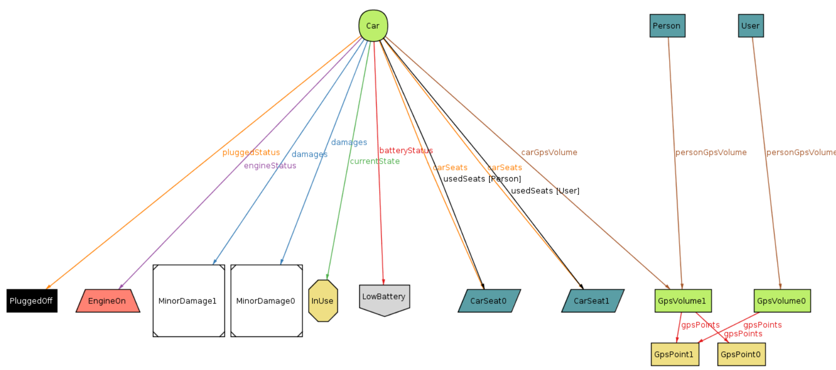

Figure 4: A Cars World

We have also shown in 5 that the execution of all the assertions have not generated counterexamples, so we can assume reasonably assume that our model is consistent.

An important aspect of our System is that a Car can be In Use, but with no person inside it. The world for this scenario is represented in Figure 6. Another interesting aspect shown in this image is that, although no one is inside the Car, its engine is still on.

Another meaningful aspect of our System is the possibility to have perfectly functioning cars whose status is Unavailable. This is surely due to some external Employee who have manually set the status of the Car for whatever reason. This world is shown in Figure 7.

```
17 commands were executed. The results are:
  #1: No counterexample found. allPersonsCantBeInDifferentCars may be valid.
  #2: No counterexample found. allMajorDamagedCarsAreUnavailable may be valid.
  #3: No counterexample found. allReservedOrAvailableCarsHaveHighBatteries may be valid.
  #4: No counterexample found. noCarInUseHaveZeroBattery may be valid.
  #5: No counterexample found. allCarWithUsedSeatsShouldBeInUse may be valid.
  #6: No counterexample found. allCarsNotInUseAndNotPluggedAndWithLowBatteryShouldBeUnavailable may be valid.
  #7: No counterexample found. noPluggedCarIsInUse may be valid.
  #8: No counterexample found. allEnginesOnAreAssociatedToInUseCars may be valid.
  #9: No counterexample found. allUsedSeatsHaveSamePositionOfCars may be valid.
  #10: Instance found. showCouldExistSomeUnavailableCarWithNoMajorDamageAndHighBattery is consistent.
  #11: Instance found. showCouldExistSomeCarWithLoweBattery is consistent.
  #12: Instance found. showCouldExistSomeAvailableCarWithMinorDamages is consistent.
  #13: Instance found. showCouldExistSomeInUseCarsWithEngineOff is consistent.
  #14: Instance found. showCouldExistSomeInUseCarsWithEngineOnAndAllPersonsOutside is consistent.
  #15: Instance found. showCouldExistSomeInUseCarsWithAllSeatsOccupiedByNonUsers is consistent.
  #16: Instance found. showMorePersonsInOneCar is consistent.
  #17: Instance found. show is consistent.
```

Figure 5: Executions of checks and predicates for Cars

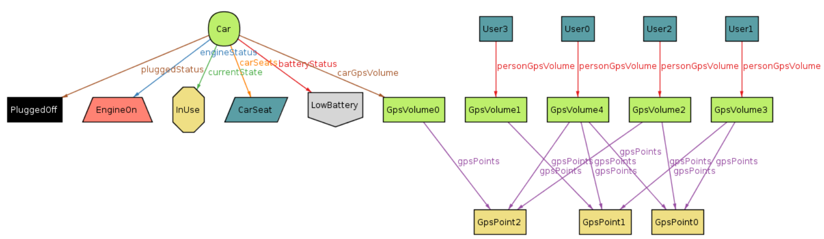

Figure 6: Used cars with no person inside

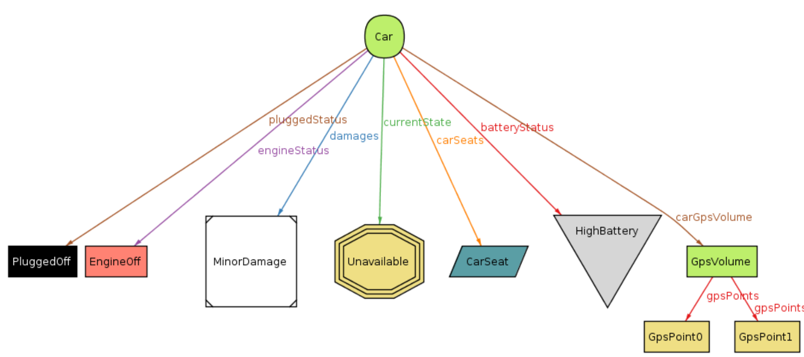

Figure 7: Unavailable functioning cars

0.1.4 Areas

```
module Areas
open Cars
open GeoUtilities

/**
  SIGNATURES
*/
abstract sig CompanyCarSlot {}
sig ParkingSlot, ChargingSlot extends CompanyCarSlot {}

abstract sig CompanyArea {
  // We assume that a CompanyArea is composed by a non empty set of Points
  // This is enough for our modelation of the world
  areaGpsPoints: some GpsPoint
}

sig ParkingArea extends CompanyArea {
  parkingSlots: set ParkingSlot,
  parkedCars: Car lone -> lone parkingSlots
}

sig ChargingArea extends ParkingArea {
  chargingSlots: some ChargingSlot,
  chargingCars: Car lone -> lone chargingSlots
}

/**
  FACTS
*/
// Trivial
fact parkingSlotsAreaAssociatedToExactlyOneArea {
  all ps: ParkingSlot | one pa: ParkingArea | ps in pa.parkingSlots
}

fact chargingSlotsAreaAssociatedToExactlyOneArea {
```

```
  all cs: ChargingSlot | one ca: ChargingArea | cs in ca.chargingSlots
}

// Areas do not overlap
fact areaPositionsAreAssociatedToExaxtlyOneCompanyArea {
//  Gps volumes for company area are predefined, so there is no way different
//  areas overlap
  all disj a1, a2: CompanyArea |
    a1.areaGpsPoints & a2.areaGpsPoints = none
}

// Parked Cars are nearby Parking Areas
fact allParkedCarsAreInsideThoseAreaPositions {
  all pa: ParkingArea, c: Car |
    c in (pa.parkedCars).(pa.parkingSlots) implies
    c.carGpsVolume.gpsPoints & pa.areaGpsPoints ≠ none
}

//Charging Cars are nearby Charging Areas
fact allChargingCarsAreInsideThoseAreaPositions {
  all ca: ChargingArea, c: Car |
    c in (ca.chargingCars).(ca.chargingSlots) implies
    c.carGpsVolume.gpsPoints & ca.areaGpsPoints ≠ none
}

// If a Car is inside an Area but not occupying a slot, it should be in use
fact allCarsInsideAreasButNotParkedOrChargingAreInUse {
  all c: Car |
    (c.carGpsVolume.gpsPoints in ParkingArea.areaGpsPoints and
     c not in
     ( (ParkingArea.parkedCars).ParkingSlot +
       (ChargingArea.chargingCars).ChargingSlot )) implies
       c.currentState = InUse
}
```

```
// I.e. a ParkingArea has always a parkingCapacity > 0
fact parkingCapacityZeroCanOnlyBeAssociatedToChargingArea {
  all p: ParkingArea | p.parkingSlots = none implies
    p in ChargingArea
}

// N.B.: Implies and not Iff bcz a car in a ParkingArea can also be Unavailable
fact carStateAvailableOrReservedImpliesCarAtOneParkingArea {
  all c: Car, pa: ParkingArea, ca: ChargingArea |
    (c.currentState = Available or c.currentState = Reserved) implies
    ( (c in (pa.parkedCars).ParkingSlot) or
      (c in (ca.parkedCars).ParkingSlot) or
      (c in (ca.chargingCars).ChargingSlot ))
}
// If a car is plugged ≤> it must be in one charging area
fact carStatePluggedIffCarInOneChargingCars {
  all c: Car | one ca: ChargingArea |
    c.pluggedStatus = PluggedOn iff c in (ca.chargingCars).(ca.chargingSlots)
}

fact carCantBeChargingAndParkedAtSameTime {
  no (ParkingArea.parkedCars).ParkingSlot &
      (ChargingArea.chargingCars).ChargingSlot
}

fact carParkedInOneParkingArea {
  all pa1, pa2: ParkingArea |
    (pa1 ≠ pa2 implies
      (pa1.parkedCars).ParkingSlot & (pa2.parkedCars).ParkingSlot = none)
}
```

```
fact carChargingInOneChargingArea {
  all ca1, ca2: ChargingArea |
    (ca1 ≠ ca2 implies
      (ca1.chargingCars).ChargingSlot &
      (ca2.chargingCars).ChargingSlot = none)
}

fact carStateInUseIfItIsNotInAParkingOrChargingSlot {
  all c: Car | c.currentState = InUse implies
    c not in ( (ParkingArea.parkedCars).ParkingSlot +
            (ChargingArea.chargingCars).ChargingSlot)
}

/**
  ASSERTS
*/
assert areaPositionsAreNotOverlapping {
  all disj ca1, ca2: CompanyArea | ca1.areaGpsPoints &
   ca2.areaGpsPoints = none
}
check areaPositionsAreNotOverlapping for 10

assert sameCarShouldNotBePluggedAtDifferentChargingArea
    {
  all c: Car | one ca: ChargingArea |
    c.pluggedStatus = PluggedOn iff
    c in (ca.chargingCars).(ca.chargingSlots)
}
check sameCarShouldNotBePluggedAtDifferentChargingArea
   for 10

assert sameCarShouldNotBeParkedAtDifferentParkingArea {
  all disj p1, p2: ParkingArea |
    (p1.parkedCars).ParkingSlot & (p2.parkedCars).
   ParkingSlot = none
}
check sameCarShouldNotBeParkedAtDifferentParkingArea
   for 10

```

```
// Bcz we assume disjoint sets
assert sameCarShouldNotBeParkedAndChargingAtSameTime {
  no (ParkingArea.parkedCars).ParkingSlot &
      (ChargingArea.chargingCars).ChargingSlot
}
check sameCarShouldNotBeParkedAndChargingAtSameTime for 10

assert carsParkedOrChargingAreNearbyThoseAreas {
  all c: Car |
    c in ( (ParkingArea.parkedCars).ParkingSlot +
        (ChargingArea.chargingCars.ChargingSlot) )
      implies
      (c.carGpsVolume.gpsPoints & ParkingArea.areaGpsPoints ≠ none)
}
check carsParkedOrChargingAreNearbyThoseAreas for 5

assert allParkingOrChargingCarsAreNotInUse {
  all c: Car | c.currentState = InUse implies
    c not in ( (ParkingArea.parkedCars).ParkingSlot +
            (ChargingArea.chargingCars).ChargingSlot)
}
check allParkingOrChargingCarsAreNotInUse for 10


/**
  PREDICATES/FUNCTIONS
*/
pred show() {
  all p: GpsPoint | p in Person.personGpsVolume.gpsPoints or p in CompanyArea.areaGpsPoints or
    p in Car.carGpsVolume.gpsPoints
  GpsVolume in (Person.personGpsVolume + Car.carGpsVolume)
  #GpsVolume > 1

  #Car > 0
  all c: Car | #c.carSeats < 3 and #c.damages < 2
  #Car.usedSeats > 0
```

```

  #Person > 0
  #(Person - User) > 0

  #CompanyArea > 0
  #(ParkingArea - ChargingArea) > 0

  #ParkingArea.parkedCars > 0
  #ChargingArea.chargingCars > 0
}
run show for 3
```

Here we define the CompanyAreas and all the things related to them.

Examples of possible worlds are shown in the following figures.

In Figure 8 we show a Car which is In Use and at the same time inside a Charging Area without occupying any of its charging slots. This does not come as a surprise: an User can still be inside an Area even if he/she is using the Car. However, we can also notice that, even if the Car is InUse, there is no Person occupying any of the seats. The only User shown in the figure has the same position of the ChargingArea (i.e. he/she is nearby it) and the same position of the Car (i.e. he/she is nearby it).

Figure 9, instead, shows a Charging Area with a Car inside it. The Car is occupying a ParkingSlot of this ChargingArea. Its status, however, is Unavailable, maybe due to the fact that it has ZeroBattery.

Adding more objects to the model, we can see how things get complicated (but still consistent). Possible worlds are shown in figures 10 and 11.

Even in this case, we can see in figure 12 how the execution of all checks has not shown any counterexample for our model.

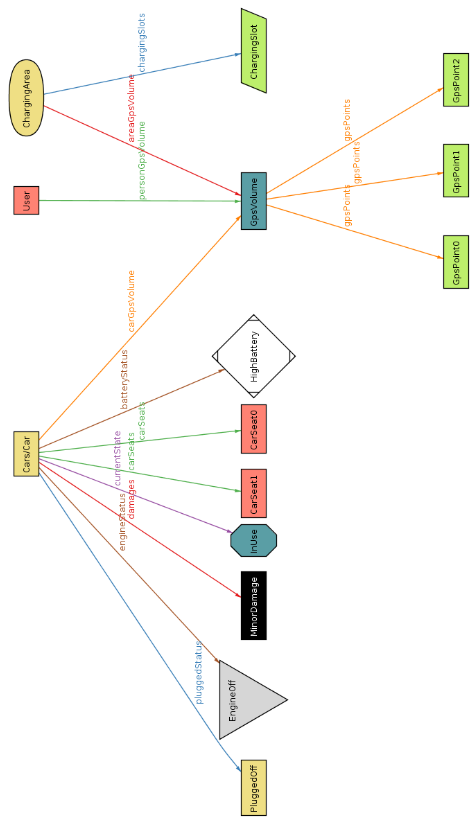

Figure 8: An Areas World

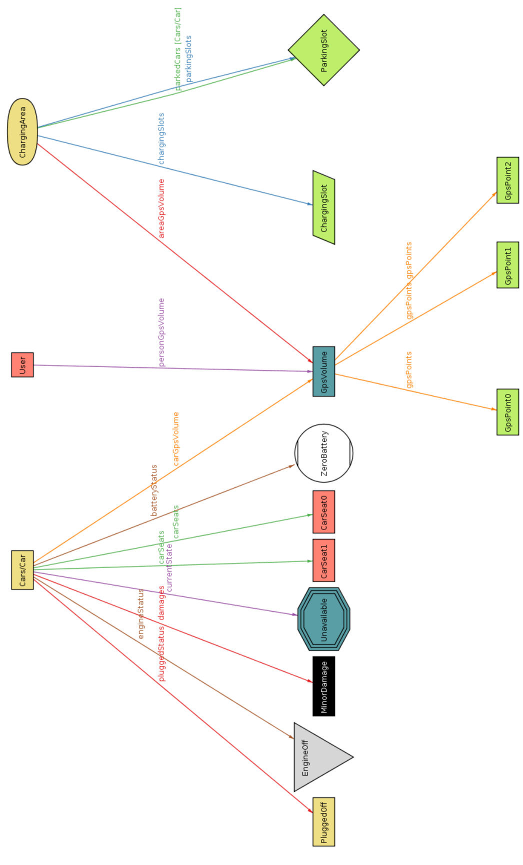

Figure 9: Another Areas World

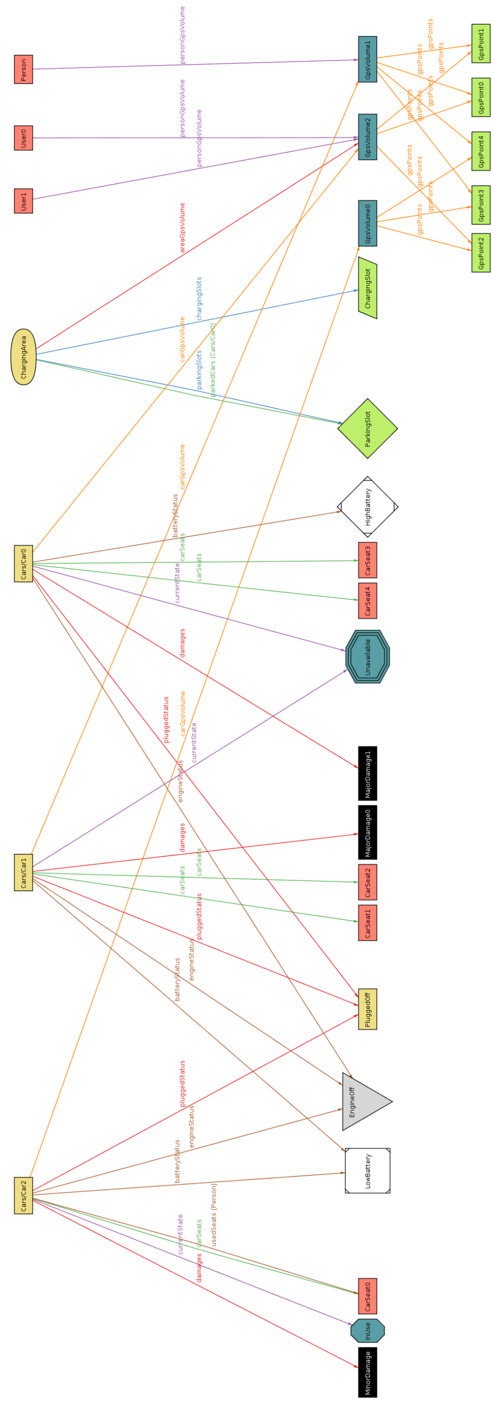

Figure 10: A more complicated Areas World

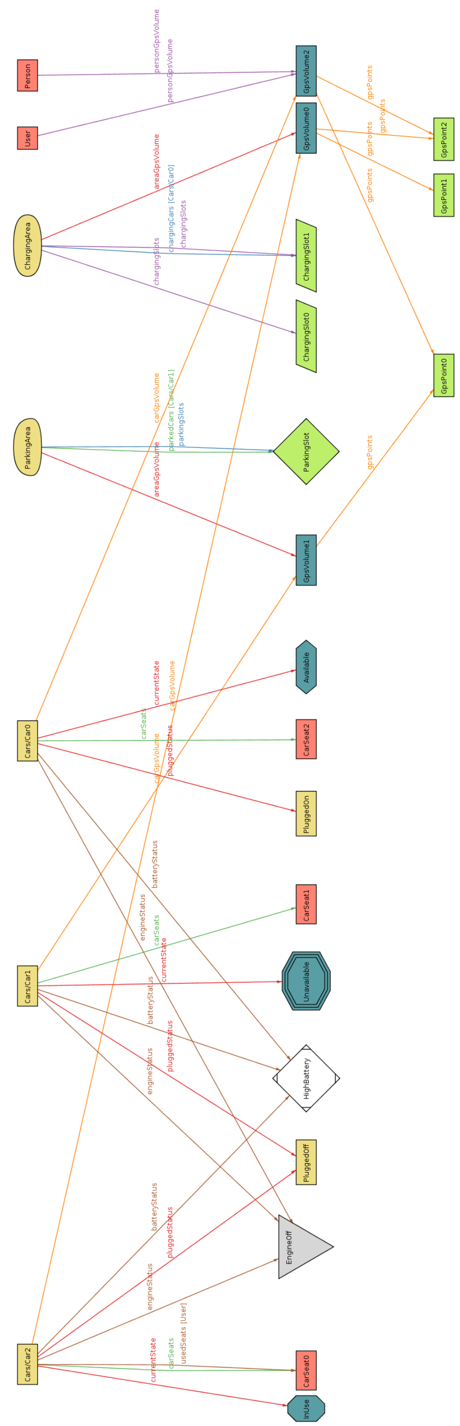

Figure 11: Another more complicated Areas World

```
7 commands were executed. The results are:
  #1: No counterexample found. areaPositionsAreNotOverlapping may be valid.
  #2: No counterexample found. sameCarShouldNotBePluggedAtDifferentChargingArea may be valid.
  #3: No counterexample found. sameCarShouldNotBeParkedAtDifferentParkingArea may be valid.
  #4: No counterexample found. sameCarShouldNotBeParkedAndChargingAtSameTime may be valid.
  #5: No counterexample found. carsParkedOrChargingAreNearbyThoseAreas may be valid.
  #6: No counterexample found. allParkingOrChargingCarsAreNotInUse may be valid.
  #7: Instance found. show is consistent.
```

Figure 12: Execution of checks and predicates for areas