open util/boolean

//Dates are expressed as the number of days from 01/01/2000 because that's for our

//needs the best way to manage them, since Alloy didn't have a date object

//\*\*\*\*\*\*\*\*\*\* SIGNATURES \*\*\*\*\*\*\*\*\*\*

sig User{

email: Stringa,

preferences: Preference, //one because the user can only have a set of preferences

schedule: seq Meeting,

owns: set OwnMean,

subscribed: set SharingSystem,

provides: set Ticket,

}{

(#subscribed>0 or #owns>0 or #provides>0)

#schedule>0

}

sig Stringa{}

//Set of preference in boolean that the user will compile when he registers on Travlendar+

sig Preference{

carshare: Bool,

bikeshare: Bool,

ownMean: Bool,

publicMean: Bool,

walking: Bool,

house: Bool,

work: Bool,

minCarbon: Bool,

}{

all p: Preference | one u: User | p in u.preferences

}

//Represents any type of meeting that the user can setup

sig Meeting{

date: Int,

startingTime: Int,

endingTime: Int,

requires: set Travel,

locate: one Region,

}{

date>0 &&

startingTime>0 &&

endingTime>startingTime

#requires>0

}

sig Region{}

//Represents any type of travel and it's always associated to a meeting

sig Travel{

date: Int,

startingTime: Int,

endingTime: Int,

mean: set MeanOfTransp,

needed: one Ticket,

associated: one Meeting,

}{

startingTime>0 && date>0

&& endingTime>startingTime && endingTime < associated.startingTime

&& date = associated.date

}

abstract sig MeanOfTransp{

ticketNeeded: Bool,

}

sig OwnMean extends MeanOfTransp{

}{

ticketNeeded=False

}

//Represents any type of public mean

sig PublicMean extends MeanOfTransp{

need: one Ticket,

}{

ticketNeeded=True

}

//Represents any type of sharing system

sig SharingSystem extends MeanOfTransp{}

{

ticketNeeded=False

}

//Represents any type of ticket for public means

sig Ticket{

seasonPass: Bool,

}

//\*\*\*\*\*\*\*\*\*\* FACTS \*\*\*\*\*\*\*\*\*\*

fact userHasTickets{

some u: User | all t: Ticket | t in u.provides

}

//Users will travel on mean they could take

fact noUnauthorizedMean{

all u: User, t: Travel | t.mean in u.owns || t.mean in u.subscribed||

t.needed in u.provides

}

fact uniqueMail {

all u1, u2: User | (u1 != u2) => u1.email != u2.email

}

//Preferences are connected to what the user owns or provides

fact consistentPreferences{

all p: Preference | one u: User | p in u.preferences && (p.carshare=False => #u.subscribed=0)

&& (p.ownMean=False => #u.owns=0)

&& (p.bikeshare=False => #u.subscribed=0)

&& (p.publicMean=False => #u.provides=0)

}

//Meetings are connected to users

fact meetingOfAUser{

all m: Meeting | one u: User | m in u.schedule.elems

}

//There are no duplicates in the sequence of meetings

fact noDuplicates{

all u: User | not u.schedule.hasDups

}

//Travels associated to a user must start at different times

fact travelsStartAtDifferentTime{

no disj t1, t2: Travel | one m: Meeting | m in t1.associated && m in t2.associated && t1.date=t2.date

&& t1.startingTime=t2.startingTime && t1.endingTime=t2.endingTime

}

fact needTicketIfPublicMean{

all t: Travel | all m: PublicMean | #t.needed>0 <=> m.ticketNeeded=True

}

//Any Travel needs at least one mean of transport

fact AtLeastOneMeanForTravel {

some m: MeanOfTransp | some tick:Ticket | all t: Travel |

m in t.mean || tick in t.needed

}

//Travel must be compatible to what the user owns or provides

fact TravelCompatibleToUser{

all t: Travel | all u: User | all i: u.schedule.inds | t in u.schedule[i].requires =>

((some m: t.mean | m in u.schedule[i].requires.mean) ||

(some tick: Ticket | tick in u.schedule[i].requires.needed))

}

fact travelAssociatedToMeeting{

all t: Travel | one m: Meeting | t in m.requires

}

fact MeanExistsOnlyIfUsed{

all m: MeanOfTransp | one t: Travel |

m in t.mean

}

fact TicketExistsOnlyIfUsed{

all tick: Ticket | one t: Travel | tick in t.needed

}

//\*\*\*\*\*\*\*\*\*\* PREDICATES \*\*\*\*\*\*\*\*\*\*

/\*\*

\* Precondition: not m in this.schedule.elems

\*/

pred User.addMeeting[u: User, m: Meeting] {

this.schedule = this.schedule.add[m]

}

/\*\*

\* Precondition: not this.schedule.isEmpty

\*/

pred User.deleteMeeting[u: User, m: Meeting] {

m in this.schedule.elems &&

not this.schedule.hasDups &&

this.schedule.lastIdxOf[m]=0

u.schedule = this.schedule.delete[this.schedule.idxOf[m]] =>

not m in u.schedule.elems

}

pred show{

#User=1

#Meeting=2

}

//\*\*\*\*\*\*\*\*\*\* ASSERTIONS \*\*\*\*\*\*\*\*\*\*

assert addChangesSchedule {

all u1, u2: User, m: Meeting | #u1.schedule = #u2.schedule && (u1.addMeeting[u1, m] =>

#u2.schedule < #u1.schedule)

}

assert deleteInverseOfAdd {

all u: User, m: Meeting, s: u.schedule | u.addMeeting[u, m] and u.deleteMeeting[u, m] =>

s = u.schedule

}

assert oneTravelAtATime{

no u: User | one disj t1, t2: Travel | all i: u.schedule.inds | t1 in u.schedule[i].requires

&& t2 in u.schedule[i].requires && t1.date=t2.date

&& (t1.endingTime>t2.startingTime || t2.endingTime>t1.startingTime)

}

//\*\*\*\*\*\*\*\*\*\* RUNS AND CHECKS \*\*\*\*\*\*\*\*\*\*

run show for 5

check oneTravelAtATime

check addChangesSchedule

run deleteMeeting for 1

run addMeeting for 1

check deleteInverseOfAdd