Métodos Formais em Engenharia de Software
Teste, 14 de Dezembro de 2024

Número: _____ Nome: _____

1. SAT *(4 points)*

A software development team with 4 members (*Ann*, *Mary*, *Peter*, and *John*) must attend virtual meetings about three software development phases (*Design*, *Coding*, *Testing*), according to the following rules.

1. All team members must attend at least one meeting.
2. All meetings must have at least two participants.
3. Mary does not want to attend the design meeting.
4. Peter and John do not want to be together in any meeting.
5. Participants in the coding meeting must attend the testing meeting.

a) Encode this problem in propositional logic. Clearly state what each variable represents and write propositional formulas that encode the above rules.

We could use a boolean variable $x_{p,m}$ for each person p and each meeting m. That variable is true iff person p attends meeting m. The above rules could be encoded in propositional logic as follows, assuming P is the set of all persons and M is the set of all meetings, and that each person and meeting is identified by the first upper-case letter:

1. $\bigwedge_{p \in P} (\bigvee_{m \in M} x_{p,m})$
2. $\bigwedge_{m \in M} ((x_{A,m} \wedge (x_{M,m} \vee x_{P,m} \vee x_{J,m})) \vee (x_{M,m} \wedge (x_{P,m} \vee x_{J,m})) \vee (x_{P,m} \wedge x_{J,m}))$
3. $\neg x_{M,D}$
4. $\bigwedge_{m \in M} \neg(x_{P,m} \wedge x_{J,m})$
5. $\bigwedge_{p \in P} (x_{p,C} \rightarrow x_{p,T})$

b) Explain how you could use a SAT solver to check the validity of the following statement: the design meeting will be attended by exactly two team members.

We could add the above constraints to the solver together with the negation of the following constraint that encodes the statement that we want to check if it is valid. If the solver returns SAT then the statement is invalid, otherwise it is valid.

$$((\neg x_{A,D} \wedge (\neg x_{M,D} \vee \neg x_{P,D} \vee \neg x_{J,D})) \vee (\neg x_{M,D} \wedge (\neg x_{P,D} \vee \neg x_{J,D})) \vee (\neg x_{P,m} \wedge \neg x_{J,m}))$$

Since we already have a constraint that forces each meeting to have at least two participants, it suffices to check the validity of the statement that specifies that there are at most two participants in the design meeting (or at least two that do not participate).

Métodos Formais em Engenharia de Software
Teste, 14 de Dezembro de 2024

Número: _____ Nome: _____

2. Structural modeling with Alloy *(4 points)*

Consider the following model of virtual meeting apps. Meetings can occur in one of two apps, and once scheduled they have a host. When the host starts the meeting they become active and participants can join to attend the meeting.

```
sig User {}

abstract sig App {}
one sig Zoomy, Xteams extends App {}

sig Meeting {
      host : lone User,
      participants : set User,
      app : one App
}
sig active in Meeting {}
```

Specify the following requirements:

a) Active meetings must have a host.

```
all m : active | some m.host
```

b) Only active meetings have participants.

```
all m : Meeting | some m.participants implies m in active
```

c) `Xteams` does not allow users to participate in two meetings.

```
all u : User | lone participants.u & app.Xteams
```

d) `Zoomy` does not allow the host of an active meeting to participate in another meeting.

```
all disj a,b : active & app.Zoomy | a.host not in b.participants
```

Métodos Formais em Engenharia de Software
Teste, 14 de Dezembro de 2024

Número: _____ Nome: _____

3. Specifying concepts with Alloy *(4 points)*

Consider the following concept of virtual meeting based on a simplified version of the previous model. The subset signature `active` and relations `host` and `participants` are now mutable, and we have the following actions: `schedule` a meeting; `start` a meeting that was previously scheduled; `end` an active meeting; and `join` or `leave` a meeting.

```
sig User {}
sig Meeting {
  var host : lone User,
  var participants : set User
}
var sig active in Meeting {}

pred schedule [m:Meeting, u:User] { ... }
pred start [m:Meeting] { ... }
pred end   [m:Meeting] { ... }
pred join  [m:Meeting, u:User] { ... }
pred leave [m:Meeting, u:User] { ... }
```

a) Specify action `end`, that terminates an active meeting, removing all its participants.

```
pred end [m : Meeting] {
     m in active
     participants' = participants - m->User
     active' = active - m
     host' = host
}
```

b) Specify the following operational principles:

1. A meeting can only be `scheduled` once.

```
all m : Meeting | always {
     (some u : User | schedule[m,u]) implies
      after always (no u : User | schedule[m,u])
}
```

2. After a meeting `ends` no one can `join` it before it `starts` again.

```
all m : Meeting | always {
     end[m] implies after (start[m] releases (no u : User | join[m,u]))
}
```

Métodos Formais em Engenharia de Software
Teste, 14 de Dezembro de 2024

Número: _____ Nome: _____

4. Specifying concepts with Why3 *(3 points)*

a) Consider the record type `stateT` to represent the abstract state of the previous concept:

```
type stateT = { ghost mutable host : fset (meeting, user) ;
                ghost mutable participants : fset (meeting, user) ;
                ghost mutable active : fset meeting }
invariant { forall m :meeting, u1 u2 :user.
                    mem (m,u1) host /\ mem (m,u2) host -> u1 = u2 }
by { host = empty ; participants = empty ; active = empty }
```

Write type invariants corresponding to the following Alloy elements:

1. The unicity constraint **`lone`** `User` in the declaration of relation `host`;
2. The abstract invariant **`always`** (**`all`** `m : active | m.host` **`in`** `m.participants`).

```
(* 1 already given above *)

(* 2 *)
  invariant  { forall m :meeting, u :user.
               mem m active /\ mem (m,u) host -> mem (m,u) participants }
```

b) Consider now the `start` action, with the following Alloy specification:

```
pred start [m : Meeting] {
      m not in active
      some m.host
      active' = active + m
      participants' = participants + m->m.host
      host' = host }
```

Complete the definition of the corresponding ghost function `start`, including the contract and the ghost code.

```
let ghost start (m : meeting) : ()
  requires { ... }
  ensures  { ... }
= ...
```

Feel free to use the following auxiliary definition:

```
  val function getHost (m :meeting) (s :fset (meeting, user)) : user
    requires { exists u :user. mem (m,u) s }
    ensures  { mem (m,result) s }
```

```
let ghost start (m : meeting) : ()
  requires { not mem m state.active }
  requires { exists u :user. mem (m,u) state.host }
  ensures  { state.host = old state.host }
  ensures  { forall u :user. mem (m,u) state.host ->
               state.participants = add (m,u) (old state.participants) }
         (* OR state.participants = add (m,getHost m state.host)
                                    (old state.participants) *) }
  ensures  { state.active = add m (old state.active) }
=
  let host = getHost m state.host in
  state.participants <- add (m,host) state.participants ;
  state.active <- add m state.active
```

Métodos Formais em Engenharia de Software
Teste, 14 de Dezembro de 2024

Número: _____ Nome: _____

5. Implementing concepts with Why3 *(5 points)*

a) Moving on to the definition of an actual implementation for the concept, consider the following additional fields added to the `stateT` record type

```
mutable host_ : MapMeeting.t user ;
mutable participants_ : MapMeeting.t (list user) ;
mutable active_ : list meeting ;
```

where `MapMeeting` results from:

```
clone fmap.MapApp as MapMeeting with type key = meeting
```

Write one or more type invariants capturing the refinement correspondence for this implementation.

```
(* Refinement correspondence *)
  invariant { forall m :meeting, u : user. mem (m,u) host <->
                MapMeeting.mem m host_ /\ host_ m = u }

  invariant { forall m :meeting, u : user. mem (m,u) participants <->
                MapMeeting.mem m participants_ /\ Mem.mem u (participants_ m) }

  invariant { forall m :meeting. mem m active <-> Mem.mem m active_ }
```

b) Complete the definition of function `start` (give only the implementation code).

```
  let start (m : meeting) : ()
  =
    let host_ = MapMeeting.find m state.host_ in
    if not MapMeeting.mem m state.participants_
      then state.participants_ <- MapMeeting.add m Nil state.participants_ ;
    let parts = MapMeeting.find m state.participants_ in
    state.participants_ <-
      MapMeeting.add m (Cons host_ parts) state.participants_ ;
    state.active_ <- Cons m state.active_
```

c) Consider an alternative implementation using only lists:

```
type stateT = { mutable host_ : list (meeting, user) ;
                mutable participants_ : list (meeting, user) ;
                mutable active_ : list meeting ;
                ghost mutable host : fset (meeting, user) ;
                ghost mutable participants : fset (meeting, user) ;
                ghost mutable active : fset meeting }
invariant { host = elements host_ }
invariant { participants = elements participants_ }
invariant { active = elements active_ }
```

Give a new definition of function `start` based on this type (give only the implementation code).

```
(* aux function *)
let rec function getHost_ (m :meeting) (l :list (meeting, user)) : user
  requires { exists u :user. Mem.mem (m,u) l }
  ensures  { Mem.mem (m,result) l }
  variant  { l }
=
  match l with
  | Nil -> absurd
  | Cons (m', u) t -> if m' = m then u else getHost_ m t
  end


let start (m : meeting) : ()
=
  let host_ = getHost_ m state.host_ in
  state.participants_ <- Cons (m, host_) state.participants_ ;
  state.active_ <- Cons m state.active_
```