Homework 7: Modeling a team sports game

Check CMSX for due date

For this programming project, you get to work in self-organized groups of up to three students if you like, but you can also do the project individually. You are not allowed to share your code, and you are not allowed to submit code generated by AI.

The Project

You are to develop a model for a sports game between two teams of the same size, like badminton, basketball, or volleyball say. Each team must have at least 1 member. There are players that want to play. An umpire collects a set of players and forms teams. When done, there is a game. After the game, the whole thing repeats until all players have played a game. There can be only one game at a time. This is modeled with the following Harmony program:

```
from game import *

const NPLAYERS = 6

thegame = Game()

def player(self):
    # Join a game
    let (team1, team2) = game_join(?thegame, self):
        assert self in (team1 | team2)

    # Play

    # Leave a game
    game_leave(?thegame, self)

spawn eternal game_umpire(?thegame)

for id in { 1 .. NPLAYERS }:
    spawn player(id)
```

In this program, there are `NPLAYERS` players that want to play a game (which should be even). Each player invokes *game_join*(*g*, *id*) where *g* is a pointer to a *game* object and *id* is a unique identifier for the player. *game_join*(*g*, *id*) returns a tuple (*team1*, *team2*), each of which are sets of

identifiers of players who were also trying to play a game. When the player is done playing, they invoke *game_leave*(*g*, *id*). After all players have left, a new game can start if there are any players who haven't played yet.

In this particular case, there are six players, and each player plays exactly one game. So there may be one game of 3x3 players, three games of 1x1 player, or two games with first 1x1 and then 2x2 players or first 2x2 and then 1x1 player. The game object should allow players who want to play more than once as well.

There is an *umpire* who selects the teams. It is an *eternal* thread (a thread that doesn't terminate) and, repeatedly, (1) waits for a sufficient number of players, (2) forms teams of those players, (3) starts the game, and (4) waits for the game to end. All this is captured in a method *game_umpire*. The umpire is eager: as soon as there are enough players, they create the largest teams possible (if there are an odd number of players, one of the players would not be selected).

Code.

You are to write three Harmony programs: `game.hny`, `teams.hny`, and `game_broken.hny`. `game.hny` should implement the following methods:

Game(): returns the initial state of a game object;

game_umpire(*g*): the code that an umpire thread runs;

game_join(*g*, *id*): player *id* wants to join the game. Returns a tuple consisting of team1 and team 2. Every player should see the same tuple of teams;

game_leave(*g*, *id*): player *id* is done and leaves the game.

Make sure you spell those methods exactly as above, or the autograder won't be able to find them.

`teams.hny` should be an extension of the program above, spawning `NPLAYERS` player threads and an eternal umpire thread. The extensions involve improving testing. The code above only tests if the player that joined is in one of the teams, but there are many other things you can test. For example, are the teams disjoint and of the same size? You can if you like (but do not have to) use the keywords **atomically**, **sequential**, **invariant**, and **finally** for this in addition to **assert**. Do not use **print** however—this is not a differential test of external behaviors. Instead, try to figure out what invariants must hold and check those. You may have to introduce additional state, similar to the test programs we have seen in class that tested for the correctness of locks and reader/writer locks, or the ClubHouse question on Prelim 2.

Finally, `game_broken.hny` should implement the same interface as *game.hny* but do it incorrectly. The bug should ideally be subtle (so it requires a good quality test program to find it), and your `teams.hny` program should work when run using the `game.hny` module but fail when run against `game_broken.hny` (using the *–m game=game_broken* flag to harmony). The teams program above should not fail when run using your `game_broken.hny` module, because it's not a very good test program.

It may be useful to know that Harmony supports the following set operators:

{}: the empty set;

len(*s*): the cardinality (size) of set *s*;

$s \mid t$: computes the union of sets s and t;

s & t: computes the intersection of sets s and t;

$s - t$: computes set difference: s minus t;

Harmony also has a handy *set* module with operations such as subset. It is *not* true in Harmony that $<$ represents the strict subset relation. (In Harmony, $<$ is a total order on Harmony values, while the subset relation is a partial order.) To test if s is a subset of t you could check if (s & t) $==$ s or, equivalently, if ($s - t$) $==$ {}.

Submitting your work

You have to submit `game.hny`, `teams.hny`, and `game_broken.hny` to CMSX. If you work in a group, submit your work as a single group in CMSX.