

Azul
Programación Declarativa

Dalianys Pérez Perera
Dayany Alfaro González
C-411

Índice

1. Ideas generales de la solución	3
1.1. Diseño y modelación	3
2. Estrategia	9
3. Simulación	12
4. Ejecución	14

1. Ideas generales de la solución

Azul como la mayoría de los juegos de mesa requieren de una mesa, o soporte físico para poder jugarse, en estos el azar es una característica muy importante además del uso de estrategias por cada uno de los jugadores. Para la reproducción del juego es fundamental una primera etapa de análisis y diseño orientado a objetos con el objetivo de representar cada una de sus partes, siendo esto posible mediante la creación de objetos dinámicos.

Al analizar las especificaciones y reglas del juego se tuvo en cuenta que los jugadores actuarían como agentes sin la intervención de un usuario y tienen conocimiento del estado del juego(ambiente) en todo momento, además de poder modificar su propio estado y el de su ambiente. A cada jugador se le asigna un *Id* y está constituido por un tablero personal conformado por la zona de preparación(*Stair*), el muro(*Wall*), la fila de penalización(*Floor*) y una puntuación parcial(*Score*). Mientras que el estado del ambiente está dado por los respectivos estados de las factorías, el centro de la mesa, la tapa de la caja y el turno del siguiente jugador.

1.1. Diseño y modelación

Sobre la base de los conceptos encontrados durante el análisis del problema, nuestra propuesta de solución agrupa los predicados implementados en los siguientes módulos:

- **environment.pl** : Contiene los predicados que modelan el transcurso de la partida a través de la definición de los conceptos del ambiente del juego vistos anteriormente. Aquí se definen los predicados dinámicos:
 - **first_player(P)**. triunfa si P es el *Id* del primer jugador de la siguiente ronda. En la primera ronda se comienza por el jugador con $Id = 1$ y en lo adelante será el primero en seleccionar azulejos del centro
 - **next_turn(P)**. triunfa si P es el *Id* del jugador que le corresponde la siguiente jugada.
 - **cant_factory(F)**. triunfa si F es la cantidad de factorías existentes en la partida actual.
 - **factory(F, B, Y, R, G, W)**. triunfa si la factoría con $Id = F$ tiene cantidades de azulejos iguales a B azules, Y amarillos, R rojos, G verdes y W blancos.
 - **center(Chip, B, Y, R, G, W)**. triunfa si el centro de la mesa tiene cantidades de azulejos iguales a B azules, Y amarillos, R rojos, G verdes y W blancos y además si la ficha de jugador inicial aún está en el centro entonces $Chip = 1$, si no $Chip = 0$.
 - **bag(N, L)**. triunfa si la bolsa tiene N azulejos y L es la lista que los contiene. Por ejemplo si la bolsa tuviera un azulejo de cada color entonces N sería igual a 5 y $L = [1, 2, 3, 4, 5]$.
 - **lid(N, L)**. triunfa si la tapa tiene N azulejos y L es la lista que los contiene.
 - **stop_play(S)**. triunfa si se cumple condición de parada del juego y $S = 1$, si el juego no ha terminado $S = 0$.

La facilidad de unificación brindada por el lenguaje nos permite consultar cada uno de los atributos anteriores en cualquier momento deseado, siempre y cuando estos estén debidamente inicializados en la base de conocimiento de Prolog. Por ejemplo si queremos saber el estado de la factoría número 2 ejecutamos `factory(2, B, Y, R, G, W)`. y las variables libres unificarán de la siguiente forma $B = 1$, $Y = 2$, $R = 1$, $G = W$, $W = 0$. indicando que tiene 1 azulejo azul, 2 amarillos y 1 rojo.

Cada uno de los atributos anteriores tiene su correspondiente predicado que modifica sus valores en la base de conocimiento, por ejemplo:

```
set_first_player(ID) :- (retract(first_player(_)), !; true),
                        assert(first_player(ID)).

set_factory(ID,B,Y,R,G,W) :- (retract(factory(ID,_,_,_,_,_)), !; true),
                             assert(factory(ID,B,Y,R,G,W)).
```

Para el caso de la bolsa(bag), el centro(center) y la tapa(lid), las modificaciones que se les pueden realizar son la adición o eliminación de un azulejo y el rellenado de la bolsa con todos los azulejos de la tapa una vez se quede vacía.

A continuación se observan los predicados que encapsulan el concepto de bolsa, que como ya se ha dicho, está representada por una lista con los colores de sus azulejos. Esta representación es muy ventajosa a la hora de extraer un azulejo al azar para poner en una factoría. La forma de resolverlo es generando un número random entre 1 y la longitud de la lista y el color que ocupe esta posición será retornado y por consiguiente eliminado de la bolsa.

```
create_bag(0, 0, 0, 0, 0, []) :- !.
create_bag(0, 0, 0, 0, W, [5|T]) :- !, W1 is W-1, create_bag(0, 0, 0, 0, W1, T).
create_bag(0, 0, 0, G, W, [4|T]) :- !, G1 is G-1, create_bag(0, 0, 0, G1, W, T).
create_bag(0, 0, R, G, W, [3|T]) :- !, R1 is R-1, create_bag(0, 0, R1, G, W, T).
create_bag(0, Y, R, G, W, [2|T]) :- !, Y1 is Y-1, create_bag(0, Y1, R, G, W, T).
create_bag(B, Y, R, G, W, [1|T]) :- B1 is B-1, create_bag(B1, Y, R, G, W, T).

init_bag() :-
    create_bag(20, 20, 20, 20, 20, L),
    assert(bag(100, L)).

remove_tile_bag(-1) :- bag(N, _), N:=0, !. %si la bolsa está vacía retorna -1
remove_tile_bag(C) :- bag(N, L), N2 is N-1, N1 is N+1, random(1, N1, I),
                    retract(bag(N, L)), remove(L, I, R, C), assert(bag(N2, R)).

fill_bag() :- bag(N1, B), retract(bag(N1, B)),
               lid(N2, L), retract(lid(N2, L)),
               concat(B, L, R), N3 is N1 + N2, %concatena la bolsa y la tapa
               assert(bag(N3,R)), assert(lid(0, [])). %la tapa está vacía ahora
```

[illegible]

- **board.pl** En este se encuentran los predicados que definen las partes del tablero de cada jugador. El muro está constituido por 25 celdas dinámicas que en conjunto forman una matriz de 5×5 . Cada celda tiene asignado el *Id* del jugador correspondiente, la fila, columna, el color de la misma y si está ocupada por un azulejo o no. El predicado `cell/5` es dinámico y para actualizar o consultar una posición en el muro se utiliza el siguiente predicado:

La representación de la zona de preparación y de la línea de penalización son similares al muro, pues para ellos también se tienen predicados dinámicos `stair/4` y `floor/5`

5

A continuación se muestran capturas de la consola

```

Factory 1: 00013   Factory 2: 00211   Factory 3: 10003   Factory 4: 12001   Factory 5: 00211
Center-> 00000 |1|
BAG-> Size: 80
LID-> Size: 0
Player:1   Score:0

-----
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|-1 -1 -2 -2 -2 -3 -3 -3
|      |      |      |      |      |      |
|      |      |      |      |      |      |
-----

Player:2   Score:0

-----
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|-1 -1 -2 -2 -2 -3 -3 -3
|      |      |      |      |      |      |
|      |      |      |      |      |      |
-----

true.

```

Figura 2: Estado inicial de una partida con 2 jugadores

```

***** END GAME *****
Factory 1: 00000   Factory 2: 00000   Factory 3: 00000   Factory 4: 00000   Factory 5: 00000
Center-> 00000
BAG-> Size: 0
LID-> Size: 59
Player:1   Score:49

-----
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|-1 -1 -2 -2 -2 -3 -3 -3
|      |      |      |      |      |      |
|      |      |      |      |      |      |
-----

Player:2   Score:60

-----
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |
|-1 -1 -2 -2 -2 -3 -3 -3
|      |      |      |      |      |      |
|      |      |      |      |      |      |
-----

WINNERS !!!!!
Player:2   Score:60
true.

```

Figura 3: Estado final de la partida con 2 jugadores

- **player.pl:** Contiene los predicados relacionados con la inicialización de un jugador y la interacción de este con el ambiente en las diferentes fases del juego.

```
init_player(ID) :-
    set_score(ID, 0),
    init_board(ID).
```

En la Fase I el jugador debe tomar todos los azulejos de un mismo color de una de las ubicaciones posibles y colocarlos en alguna de las filas de su zona de preparación o en el suelo. Para la realización de estas acciones se definió el predicado `pick/1`.

```
pick(ID) :-
    get_moves(ID, All_moves),
    strategy(ID, All_moves, Source, Color, Amount, Stair, Chip),
    update_environment(Source, Color, Chip),
    place_chip(ID, Chip),
    place_colors(ID, Stair, Color, Amount).
```

- `get_moves(ID, All_moves)`: triunfa si `All_moves` es la lista que contiene todas los movimientos factibles a realizar por el jugador `ID`.
- `strategy(ID, All_moves, Source, Color, Amount, Stair, Chip)`: triunfa si de todos los movimientos factibles para el jugador `ID` contenidos en `All_moves` se escoge mover de la ubicación `Source` una cantidad `Amount` de azulejos de color `Color` hacia `Stair` con la presencia (`Chip = 1`) o no (`Chip = 0`) de la ficha de jugador inicial.
- `update_environment(Source, Color, Chip)`: triunfa si
 - `Source = 0`: se eliminan del centro de la mesa los azulejos de color `Color` y si `Chip = 1` se elimina también la ficha de jugador inicial.
 - `Source ≠ 0`: se eliminan de la factoría `Source` los azulejos de color `Color`.
- `place_chip(ID, Chip)` triunfa si
 - `Chip = 0`.
 - `Chip = 1`: si hay espacio en el suelo del jugador `ID` se coloca allí la ficha de jugador inicial. El predicado `first_player/1` es actualizado para la siguiente ronda.
- `place_colors(ID, Stair, Color, Amount)` : triunfa si
 - `Stair = 0`: se coloca en el espacio que esté disponible del suelo mediante el predicado `update_floor/4` lo que se pueda de los `Amount` azulejos de color `Color` y si hay algunos que no caben se colocan en la tapa de la caja mediante el predicado `update_lid/4`.
 - `Stair ≠ 0`: se colocan en la fila `Stair` la cantidad de `Amount` que sea posible de azulejos de color `Color` mediante el predicado `update_stair/5`, aquellos que no quepan se colocan en el suelo mientras se pueda mediante el predicado `update_floor/4` y si hay algún azulejo que no fue posible colocar en el suelo es enviado a la tapa de la caja mediante el predicado `update_lid/4`.

En la Fase II el jugador transporta un azulejo de cada una de las filas completadas al muro y por cada uno anota puntos, así como pierde puntos por cada azulejo presente en el suelo. Para simular estas acciones se definió el predicado `build_wall/1`.

```
build_wall(ID) :-
    ((setof((Stair, Color),
        (stair(ID, Stair, Stair, Color), Color=\=0),Stairs),
        build_and_clean(ID, Stairs)),!,true),
    floor_penalty(ID),
    clean_floor(ID).

build_and_clean(_, []) :- !.
build_and_clean(ID, [(Stair, Color)|Stairs]) :-
    set_value_wall(ID, Stair, _, Color, 1),
    calculate_points_horizontal(ID,Stair,Color,Horizontals),
    calculate_points_vertical(ID,Stair,Color,Verticals),
    update_score(ID,Horizontals,Verticals),
    clean_stair(ID, Stair, Color),
    build_and_clean(ID, Stairs).
```

- `calculate_points_horizontal(ID,Stair,Color,Horizontals)`: triunfa si `Horizontals` es la cantidad de azulejos directamente conectados horizontalmente a la posición del color `Color` en la fila `Stair` del muro del jugador `ID`.
 - `calculate_points_vertical(ID,Stair,Color,Verticals)`: triunfa si `Verticals` es la cantidad de azulejos directamente conectados verticalmente a la posición del color `Color` de la fila `Stair` del muro del jugador `ID`.
 - `update_score(ID,Horizontals,Verticals)`: triunfa si se actualiza el predicado `player_score/2` para el jugador `ID` según las cantidades de `Horizontals` y `Verticals`.
 - `clean_stair(ID, Stair, Color)`: triunfa si los azulejos de las filas completas de la zona de preparación que no son colocados en el muro son colocados en la tapa de la caja mediante el predicado `update_lid/4`.
 - `floor_penalty(ID)`: triunfa si la puntuación del jugador `ID` es actualizada según los azulejos contenidos en su suelo.
 - `clean_floor(ID)`: triunfa si todos los azulejos colocados en el suelo del jugador `ID` son agregados a la tapa de la caja.
- **punctuation.pl** Contiene todos los predicados relacionados con mantener la puntuación de un jugador.

Se define el pedicado dinámico `player_score(ID,Score)` que triunfa si `Score` es la puntuación del jugador `ID`.

```
set_score(ID, SCORE) :-
```



```

SCORE=<0, !,
( retract(player_score(ID, _)), !
; true
),
assert(player_score(ID, 0)), !.
set_score(ID, SCORE) :-
SCORE>0,
( retract(player_score(ID, _)), !
; true
),
assert(player_score(ID, SCORE)).

```

Aquí también encontramos los predicados auxiliares necesarios para determinar en cuánto varía la puntuación de un jugador, según las reglas del juego, tanto en la Fase II como al final de la partida, así como los predicados para reflejar dichas variaciones en `player_score/2`.

- **strategy.pl** Contiene los predicados que definen la estrategia de selección que va a seguir el jugador. En la sección 2 se describe detalladamente cuál va a ser dicha estrategia.
- **utils.pl** Contiene predicados auxiliares para el trabajo con listas y también imprimir en consola la salida de la simulación.
- **simulation.pl** Es el archivo principal que se ejecuta, pues contiene los predicados referentes a la ejecución de las tres fases. En la sección 3 se explica con más detenimiento.

2. Estrategia

Cada jugador tiene que decidir qué azulejos va a tomar, de dónde los va a tomar y dónde los va a colocar, para esto se implementó una estrategia de selección que se va a describir a continuación.

Como punto de partida es necesario tener un conjunto de todos los movimientos factibles a realizar, donde por movimiento factible se entiende que sea válido colocar en la fila seleccionada (o el suelo) los azulejos de un determinado color tomados de una de las fábricas o el centro de la mesa. Para obtener esto se crearon todas las combinaciones de movimientos posibles, o sea, tanto válidos como inválidos y se filtraron mediante la evaluación del predicado `factible_move` para determinar si eran factibles o no.

Sea `All_moves` el conjunto de movimientos factibles para el jugador `ID`, va a estar representado por una lista de tuplas con el formato `[(Source, Color, Amount, Stair, Chip),...]` que indica que se toman `Amount` fichas de color `Color` (1,2,3,4,5 según el color) desde `Source` (0,1... k) si son del suelo o de la fábrica k respectivamente) hacia la fila `Stair` y `Chip` va a ser 0 o 1 en dependencia de si se tomó o no la ficha de jugador inicial. Si `All_moves` $\neq \emptyset$ se va a seleccionar un movimiento en el siguiente orden:

1.
 - Seleccionar de All_moves aquellos movimientos que al realizarse completen exactamente una fila y con estos formar el conjunto Moves.
 - Si el jugador no lleva la delantera en la puntuación tampoco van a pertenecer a Moves aquellos movimientos que completarían una fila en el muro en la fase 2 y, por tanto, finalizarían la partida.
 - Si Moves $\neq \emptyset$:
 - Ordenar Moves de menor a mayor según la cantidad de azulejos directamente conectados en línea recta en la fila y/o columna que tiene en el muro el color que se completaría.
 - Seleccionar como movimiento a realizar el último elemento de Moves y parar de buscar, con lo que se busca maximizar el número de puntos a obtener en la fase de revestir el muro.

```

get_moves_overfill(ID, All_moves, K, Moves) :-
    setof((Ady, Src, Clr, Amnt, St, Ch),
        Free^(member((Src, Clr, Amnt, St, Ch), All_moves),
            get_free_space(ID, St, Free), Amnt-Free=:K,
            get_adyacents(ID, St, Clr, Ady),
            (is_not_game_move(ID, St, Clr);is_winning(ID))),
        Moves).

strategy(ID, All_moves, Source, Color, Amount, Stair, Chip) :-
    get_moves_overfill(ID, All_moves, 0, Moves),
    last(Moves,
        (_, Source, Color, Amount, Stair, Chip)), !.

```

2.
 - Seleccionar de All_Moves aquellos movimientos que al realizarse completen una fila y sobre exactamente un azulejo y con estos formar el conjunto Moves.
 - Si el jugador no lleva la delantera en la puntuación tampoco van a pertenecer a Moves aquellos movimientos que completarían una fila en el muro en la fase 2 y, por tanto, finalizarían la partida.
 - Si Moves $\neq \emptyset$:
 - Ordenar Moves de menor a mayor según la cantidad de azulejos directamente conectados en línea recta en la fila y/o columna que tiene en el muro el color que se completaría.
 - Seleccionar como movimiento a realizar el último elemento de Moves y parar de buscar.

```

strategy(ID, All_moves, Source, Color, Amount, Stair, Chip) :-
    get_moves_overfill(ID, All_moves, 1, Moves),
    last(Moves,
        (_, Source, Color, Amount, Stair, Chip)), !.

```

3.
 - Seleccionar de All_Moves aquellos movimientos que al realizarse no alcancen a completar exactamente una fila y además esta ya contenga al menos un azulejo, y con estos formar el conjunto Moves.
 - Si Moves $\neq \emptyset$:
 - Ordenar Moves de menor a mayor según la cantidad de espacios vacíos que quedarían en la fila al colocar los azulejos.
 - Seleccionar como movimiento a realizar el primer elemento de Moves y parar de buscar.

```
get_moves_incomplete_to_nonempty(ID, All_moves, Moves) :-
    setof((Incomplete, Src, Clr, Amnt, St, Ch),
        Free^(member((Src, Clr, Amnt, St, Ch), All_moves),
            get_free_space(ID, St, Free), Incomplete is Free-Amnt,
            Incomplete>0, stair(ID, St, 1, Clr)),
        Moves).
```

```
strategy(ID, All_moves, Source, Color, Amount, Stair, Chip) :-
    get_moves_incomplete_to_nonempty(ID, All_moves,
        [ (_, Source, Color, Amount, Stair, Chip) | _ ]), !.
```

4.
 - Seleccionar de All_Moves aquellos movimientos que al realizarse no alcancen a completar exactamente una fila y con estos formar el conjunto Moves.
 - Si Moves $\neq \emptyset$:
 - Ordenar Moves de menor a mayor según la cantidad de espacios vacíos que quedarían en la fila al colocar los azulejos.
 - Seleccionar como movimiento a realizar el primer elemento de Moves y parar de buscar.

```
get_moves_incomplete(ID, All_moves, Moves) :-
    setof((Incomplete, Src, Clr, Amnt, St, Ch),
        Free^(member((Src, Clr, Amnt, St, Ch), All_moves),
            get_free_space(ID, St, Free), Incomplete is Free-Amnt,
            Incomplete>0), Moves).
```

```
strategy(ID, All_moves, Source, Color, Amount, Stair, Chip) :-
    get_moves_incomplete(ID, All_moves,
        [ (_, Source, Color, Amount, Stair, Chip) | _ ]), !.
```

5.
 - Seleccionar de All_Moves aquellos movimientos que no provocan que se acabe el juego al llenar una fila del muro en la siguiente fase (Llegado a este punto estos movimientos se rechazaron anteriormente porque el jugador no contaba con la mayor puntuación, por tanto ahora tampoco sería conveniente seleccionarlos, mientras que el resto de los movimientos tras haber sido descartados por las variantes anteriores solo pueden ser movimientos que completen una fila y sobre

una cantidad mayor que 1 de azulejos que irían al suelo o que no se puedan colocar en ninguna fila y vayan directamente al suelo) y con estos formar el conjunto Moves.

- Si $Moves \neq \emptyset$:
 - Ordenar Moves de menor a mayor según la cantidad de azulejos que serían enviados al suelo.
 - Seleccionar como movimiento a realizar el primer elemento de F_5 y parar de buscar.

```
strategy(ID, All_moves, Source, Color, Amount, Stair, Chip) :-
    setof((Extra, Src, Clr, Amnt, St, Ch),
        Free^(member((Src, Clr, Amnt, St, Ch), All_moves),
            get_free_space(ID, St, Free), Extra is Amnt-Free,
            is_not_game_move(ID, St, Clr)),
        [ (_, Source, Color, Amount, Stair, Chip)|_]), !.
```

6.
 - Alcanzar este punto significa que todos los movimientos en All_Moves son aquellos que provocan el fin del juego al revestir el muro o que colocan las fichas directamente en el suelo
 - Ordenar All_Moves de menor a mayor según la cantidad de azulejos que serían enviados al suelo.
 - Seleccionar como movimiento a realizar el primer elemento de All_Moves.

```
strategy(ID, All_moves, Source, Color, Amount, Stair, Chip) :-
    setof((Extra, Src, Clr, Amnt, St, Ch),
        Free^(member((Src, Clr, Amnt, St, Ch), All_moves),
            get_free_space(ID, St, Free), Extra is Amnt-Free),
        [ (_, Source, Color, Amount, Stair, Chip)|_]).
```

3. Simulación

Una partida de *Azul* consta de un número indeterminado de rondas hasta que se cumpla alguna de las condiciones de finalización, estas pueden ser que al menos un jugador complete una línea en su muro o que se acaben los azulejos y ya no sea posible preparar una nueva ronda. Cada ronda consta a su vez de tres fases:

1. Selección de azulejos(Oferta de Factoría)
2. Revestimiento del muro(Alicatado de la pared)
3. Mantenimiento(Preparación de la siguiente ronda)

A modo de resumen del flujo de una partida en el juego, se observan en la figura 4 las transiciones entre las fases y las acciones correspondientes a cada una de ellas.

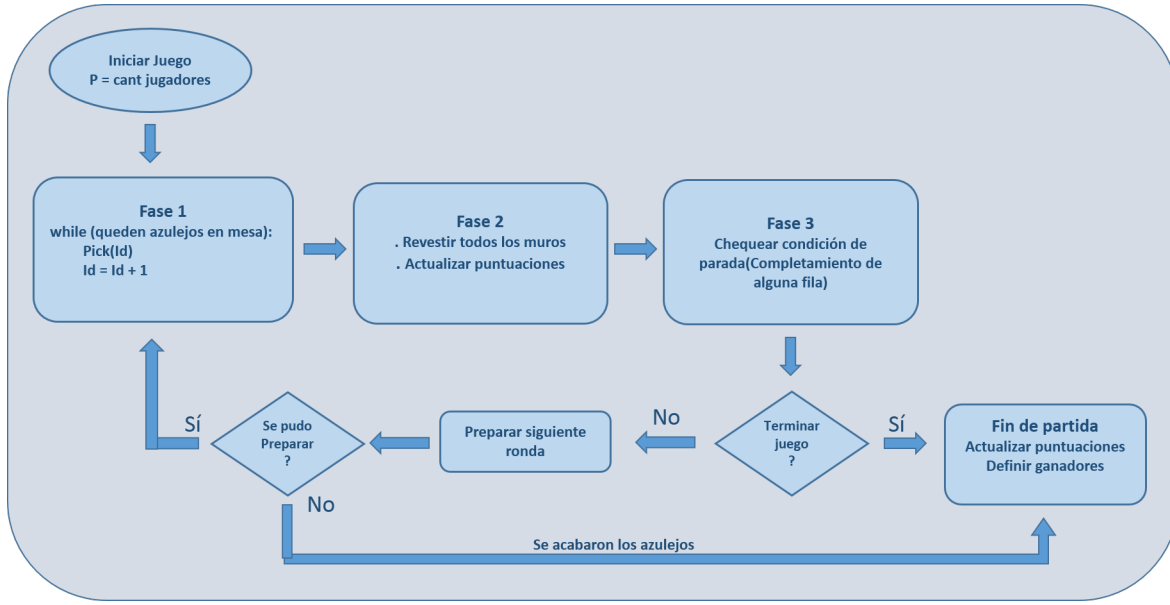


Figura 4:

A continuación se ofrece la implementación del módulo `simulation.pl` en el cual se encuentran definidos los predicados `fase_i(X)` para $i = 1, 2, 3$ y el predicado `simulate(X)`. Una vez inicializado el juego con todas sus componentes y jugadores la simulación comienza al ejecutar `simulate(1)` indicando que la próxima fase a ejecutar será la fase 1. Notar que las primeras tres cláusulas del predicado `simulate` en esencia cumplen con `simulate(I) :- faseI(X), simulate(X)`, pues X unifica con el número de la próxima fase que debe ocurrir en la partida.

```

fase1(2) :- finish_fase1(), !.
fase1(1) :- next_turn(Next), print_table(), pick(Next), print_board(Next),
            update_next_turn(), !.
fase1(5).

```

```

fase2(3) :- !, cant_players(Cant), update_all_walls(Cant).
fase2(6).

```

```

fase3(4) :- check_end_play(), !, print_end().
fase3(1) :- prepare_next_round(), !.
fase3(4) :- print_end_bag().

```

```

simulate(1) :- fase1(X), simulate(X).
simulate(2) :- printText('END_FASE_1\n', red), fase2(X), simulate(X).
simulate(3) :- fase3(X), simulate(X).
simulate(4) :- cant_players(C), winners(C, W, S), print_play_state(), printText('\n_WINN
print_winners(W, S).

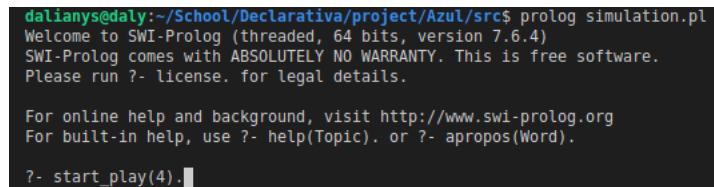
```

```
simulate(5) :- printText('(Error_in_FASE_1)', red).
simulate(6) :- printText('(Error_in_FASE_2)', red).
```

Una vez llegado a la cláusula `simulate(4)` es porque se cumplió una de las condiciones de parada, ya sea que triunfó la primera cláusula del predicado `fase3(4)` (algún jugador completó una línea) o porque falló `fase3(1)` al no poder prepararse la siguiente ronda. Estando en este punto se procede a actualizar las puntuaciones de todos los jugadores para definir aquellos que resulten vencedores de la partida.

4. Ejecución

Para correr el programa es necesario estar situado sobre la carpeta `/src` y en consola ejecutar los siguientes comandos:



```
dalianys@daly:~/School/Declarativa/project/Azul/src$ prolog simulation.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- start_play(4).
```

Figura 5:

El predicado `start_play(C)` recibe la cantidad de jugadores de la partida.