Ingeniería Informática, 4º Curso
Ingeniería del Conocimiento
Práctica 1: Sistemas de producción

García García, Alba María 100346091@alumnos.uc3m.es
Martínez Castillo, Irene 100346051@alumnos.uc3m.es
Leganés Grupo 83

22 Noviembre 2018

Índice

1. Introducción

En este documento se explica la **implementación de un Sistema de Producción capaz de realizar partidas automáticas** de una versión simplificada **del juego de mesa *Agricola*** en su versión familiar, mediante el lenguaje de programación CLIPS.

La **estructura** de este documento se divide en: un manual técnico donde se explica en detalle la ontología y las reglas utilizadas en la implementación, un manual de usuario, una descripción de las pruebas realizadas para la comprobación del correcto funcionamiento del sistema y un último apartado con conclusiones técnicas y personales.

2. Manual técnico

Hemos dividido el manual técnico en dos secciones fundamentales: la ontología de nuestro Sistema Basado en el Conocimiento y las reglas para hacer que funcione. En el caso del primero, utilizaremos diagramas de clases para facilitar su comprensión y para el segundo, lo dividiremos en secciones para estructurar el contenido lo mejor posible.

2.1. Ontología

A la hora de pensar en una ontología tuvimos en cuenta dos facetas de la misma: por un lado, una descripción del conocimiento mediante las relaciones de jerarquía, que son las únicas que permite CLIPS y por otro lado, el resto de relaciones que hemos encontrado.

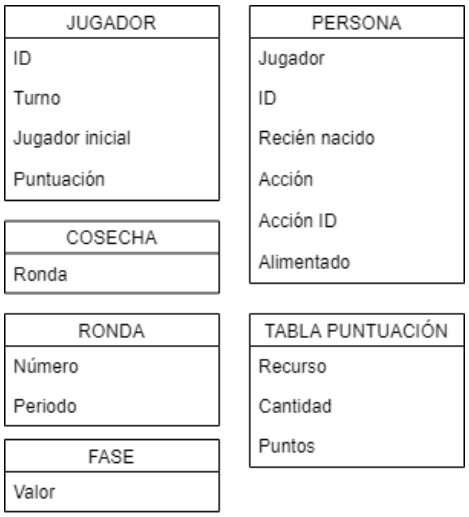

Figura 1: Clases sin jerarquía de nuestra ontología.

El concepto principal de *Agricola* son los `JUGADORES`, puesto que todo el juego gira en torno a las acciones que realizan. Para diferenciar cada uno de ellos, les hemos añadido un atributo identificador de tipo `INTEGER` (`ID`). Además, son necesarios otros atributos para indicar si es su turno o no, y si son el jugador inicial o no, ambos de tipo `SYMBOL` con valores permitidos `TRUE/FALSE`, además de un contador para su puntuación final de tipo `INTEGER`.

Cada uno de los jugadores posee diferentes `PERSONAS`. En el estado inicial cada `JUGADOR` comienza solamente con dos, pero a lo largo del juego pueden llegar a expandirse hasta un total de cinco. Cada `PERSONA`, por tanto, tiene un atributo `Jugador` con el mismo identificador que el jugador al que pertenece. Además, cada persona está diferenciada por su propio `ID`, de tipo `INTEGER`. Las personas pueden ser `Recién-nacidos`, cosa que se representa con otro atributo `SYMBOL`, cuyos posibles valores son `TRUE/FALSE`. Por último, cada `PERSONA` puede realizar una `ACCIÓN` por ronda, por lo que debemos añadirle un atributo que refleje el nombre de la carta de donde procede la acción (`Acción`) y el identificador de la acción dentro de la carta (`Acción-ID`). Por último, debemos saber si la `PERSONA` ya ha sido alimentada con `Alimentado`, un atributo `SYMBOL` con valores `TRUE/FALSE`.

También necesitamos TABLA-PUNTUACIÓN para poder asignar y relacionar los puntos necesarios por los recursos, cartas y terrenos que cada jugador tiene al final de la partida. Así, con el atributo Recurso indicamos qué elemento estamos evaluando y con Cantidad, el número de unidades del elemento que necesitamos para conseguir una cantidad de puntos representada por Puntos.

Los siguientes conceptos a identificar para poder simular una partida de *Agricola* son RONDA, COSECHA y FASE. El primero de estos conceptos sirve para identificar cada una de las 14 rondas de las que se compone el juego, cada una asociada a un Periodo. Cada ronda del juego se compone de diferentes FASES y en cada una de ellas se pueden hacer diferentes acciones. Estas fases serán explicadas más adelante en el documento. En cuanto a la COSECHA, su atributo Ronda indica en qué rondas se produce esa fase especial.

Ligado al concepto PERSONA encontramos el concepto ACCIÓN, que representa todas los posibles acciones que puede hacer cada persona dentro del juego, dependiendo de la FASE en la que se encuentran. Cada ACCIÓN consiste de un Nombre y un ID diferente. Hay algunas acciones que pueden consistir en dos acciones diferentes que deben ejecutarse de forma simultánea. Estas serán representadas con el atributo Conjunta con valor TRUE. Además, las acciones poseen el atributo Ocupada, para indicar si están siendo realizadas o no por alguna PERSONA en la ronda actual. Dentro de ACCIÓN, se definen las siguientes subclases:

- INICIO-DE-RONDA. Acciones que se ejecutan en la fase PRINCIPIO-DE-RONDA. En el caso de esta versión de *Agricola*, sirve para representar únicamente la acción "Pozo". Contiene atributos Contador y Recogido para saber las rondas en la que seguirá teniendo efecto la acción e para identificar cuándo se han recogido los recursos, respectivamente.
- RECOGER-RECURSO. Esta acción representa el hecho de recoger los recursos que se encuentran en el tablero, así como el de reponer al principio de cada ronda dichos recursos (ya sea en una CARTA-DE-TABLERO o en una CARTA-DE-RONDA). Los atributos de esta clase son: Recurso, que puede tener como valor cualquier nombre de los posibles recursos del juego; Total, para indicar la cantidad total de recursos acumulados; Ronda, que indica el número de recursos a añadir por cada ronda; y Repuesto, para indicar si hemos repuesto esos recursos en la ronda actual o aún tenemos que añadirlos.
- INTERCAMBIAR-RECURSO. Se trata de acciones que intercambian algún tipo de recurso por comida mediante adquisiciones mayores. Por tanto, los atributos de esta clase son: Recurso y Cantidad, que indican el recurso y cantidad a cambiar; Comida, que indica las unidades de comida a obtener; así como Hornear pan, que informa sobre qué si la acción se considera *hornear pan*, en caso negativo vale 0 y en caso afirmativo, el ID de la misma (un entero del 1 al 6).
- HORNEAR-PAN. Representa la acción de *hornear pan*, pero en lugar de a través de una adquisición mayor, usando de una carta de ronda. Esta subclase solamente tiene un atributo Carta, que incluye uno de los seis identificadores de las adquisiciones mayores que permiten hornear pan.
- ARAR-CAMPO. Habilita un terreno vacío para poder sembrar vegetales en futuras rondas, es decir, lo convierte en un CAMPO. Posee el atributo id-terreno para indicar qué terreno se va a arar.
- SEMBRAR. Coloca vegetales en los campos arados, ya sean estos cereales u hortalizas. Necesita de un Vegetal (con posibles valores CEREAL u HORTALIZA) y un id-terreno para indicar qué campo va a sembrarse.
- CONSTRUIR-VALLA. Delimita los terrenos mediante vallas para poder colocar recursos de tipo animal y formar pastos. Necesita también el atributo id-terreno, para saber en qué pasto construir la valla (en el caso de los pastos dobles, se utiliza su ID no su ID-extra).

- `CONSTRUIR-ESTABLO`. Coloca establos en pastos preexistentes para ampliar la capacidad máxima de animales. Al igual que todas las acciones de construcción, necesita `id-terreno`; cuyo uso es el mismo que en la acción anterior.
- `CONSTRUIR-HABITACIÓN`. Amplia el número de habitaciones de cada jugador, para poder incrementar el número de animales de compañía y de personas. Necesita también un `id-terreno`, para saber en qué terreno vacío construir la habitación.
- `REFORMAR`. Mejora todas las habitaciones de un jugador simultáneamente. No es posible reformar si no se dispone de los recursos suficientes para reformar todas las habitaciones. No necesita ningún atributo extra, porque no se puede elegir qué material utilizar en la reforma, viene predefinido.
- `AMPLIAR-FAMILIA`. Crea una nueva persona que formará parte de la familia del jugador. Esta acción puede requerir de una habitación disponible para alojar al recién nacido o no. Por esta razón, el único atributo necesario es `Habitación`, con posibles valores `TRUE-FALSE` para indicar si se necesita o no dicha habitación.
- `OBTENER-ADQUISICIÓN-MAYOR`. Permite que un jugador pueda conseguir una carta de adquisición mayor. Para saber qué adquisición mayor se quiere obtener usaremos el atributo `Carta`.
- `CAMBIAR-JUGADOR-INICIAL`. Cambia de orden el turno de los jugadores en la ronda actual comprobando cuál de los dos es el jugador inicial. No necesita ningún atributo extra porque toda la información que necesita se encuentra en la clase `JUGADOR`.

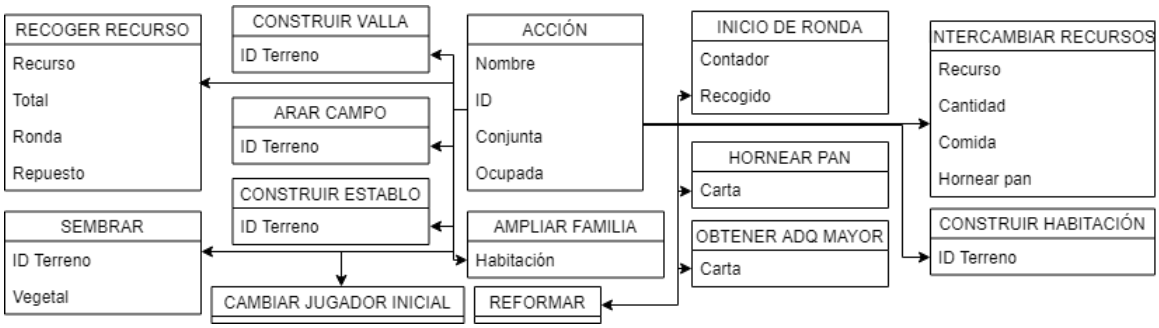

Figura 2: Jerarquía de la clase `ACCIÓN`.

Cada una de estas `ACCIONES` está relacionada con una `CARTA`, que es el siguiente concepto a definir. Cada `CARTA` se identifica mediante el atributo `Nombre` y tiene un atributo `Jugador` para poder saber quién puede utilizar esa carta. Este atributo puede tomar 4 valores: 0 cuando no la puede usar nadie, 1 y 2 cuando la pueden usar el jugador 1 y el 2, respectivamente y 3 cuando la pueden utilizar ambos jugadores. Hay `CARTAS` con diferentes características, que podemos definir como subclases:

- `CARTAS DE TABLERO`. Son las cartas con las que comienza el juego. Reciben este nombre por ser parte del tablero de juego.
- `CARTAS DE RONDA`. Se trata de las cartas que se destapan por periodos de manera aleatoria para el uso de todos los jugadores al principio de cada ronda. Estas cartas tienen un atributo `Periodo` para saber en qué periodo pueden entrar al juego.
- `CARTAS DE MENDICIDAD`. Ayudan a los jugadores a alimentar a su familia en la fase de `ALIMENTACIÓN` si no disponen de comida necesaria. No necesita ningún atributo extra porque no tienen ninguna característica particular entre ellas, ya que funcionan todas igual (mismo nombre, efectos y puntos finales).

- `CARTAS DE ADQUISICIÓN MAYOR`. Estas cartas tienen un mayor número de atributos para representar todas sus características. Estos son: `Hornear pan`, que se trata de un identificador para aquellas cartas que permiten *hornear pan*; `Hogar`, que indica si la carta es un hogar o no (veremos por qué es necesario en la sección de *Reglas*); `Puntos`, que es el número de puntos que otorga la carta por haberla comprado; `Recurso puntos extra`, que se trata del recurso que se tiene en cuenta para calcular los puntos extra; `Cantidad puntos extra`, que indica la cantidad del recurso anterior para obtener 1, 2 o 3 puntos extra (según el índice); y `Puntuada`, para saber si ya ha sido tenida en cuenta en la puntuación final.

  Las adquisiciones mayores a su vez pueden necesitar uno o dos recursos para comprarlas, por lo que hemos identificado dos subclases para poder almacenar esta información.

  - `ADQ-MAYOR-1-RECURSO`. Contiene los *slots* `Recurso` y `Cantidad` para guardar la información del único recurso que necesita.
  - `ADQ-MAYOR-2-RECURSOS`. De igual manera, este tipo de adquisición mayor tendrá *slots* suficientes para dos recursos.

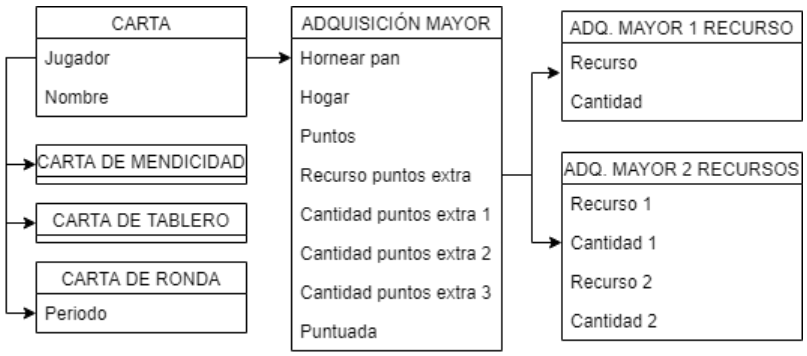

Figura 3: Jerarquía de la clase `CARTA`.

Otro concepto importante para simular las partidas de *Agricola* son los `TERRENOS`. Cada terreno simula una celda del tablero de cada jugador normalmente. La única excepción que encontramos en esta afirmación son los pastos, ya que pueden estar formados de hasta dos celdas. Esto significa que tendremos un máximo de 15 terrenos (diferenciados entre ellos con atributo `id` de tipo `INTEGER`) por jugador a su vez asociados a cada `JUGADOR` con un atributo homónimo de tipo `INTEGER`.

Los `TERRENOS` a su vez se pueden clasificar en:

- `TERRENO-VACIO`. Es el concepto más sencillo de `TERRENO`. Simboliza una celda de la granja del jugador sin ninguna especialización ni característica especial.
- `HABITACIÓN`. Identifican a los terrenos donde viven las personas que controla el jugador. Está compuesta por el atributo `Material` con posibles valores `Madera, Adobe o Piedra` y el `Animal`, que representa el posible animal de compañía que puede habitar en la habitación. Los jugadores comienzan con dos instancias de esta clase, con `Material = Madera` y `Animal = NULL`.
- `CAMPO`. Son los terrenos que permiten sembrar y cultivar `Vegetales`, pudiendo ser estos `CEREALES` u `HORTALIZAS`. Cada campo tiene además un atributo `Cantidad`, tipo `INTEGER`, con el número de elementos del recurso que posee y un atributo `Recolectado`, para poder tener contabilizados los campos que han sido recolectados de los que no en la fase de `RECOLECCIÓN`.

- `PASTO`. Son terrenos que sirven para acoger animales, por eso `Animal` es el primer atributo de esta subclase, de tipo `SYMBOL`. `Animal` puede tener los valores `Oveja, Vaca, Jabalí`, es decir, el tipo de animal que guarda el pasto. Los `PASTOS` pueden estar vallados, así como tener establos para duplicar la capacidad máxima permitida en cada pasto. Estos valores están definidos por atributos `Vallas` y `Establos`, respectivamente. Cabe destacar que los pastos pueden ser dobles, por eso poseen el atributo `id-extra`, de tipo `INTEGER`, que indica el ID de la segunda celda en caso que hiciera falta.

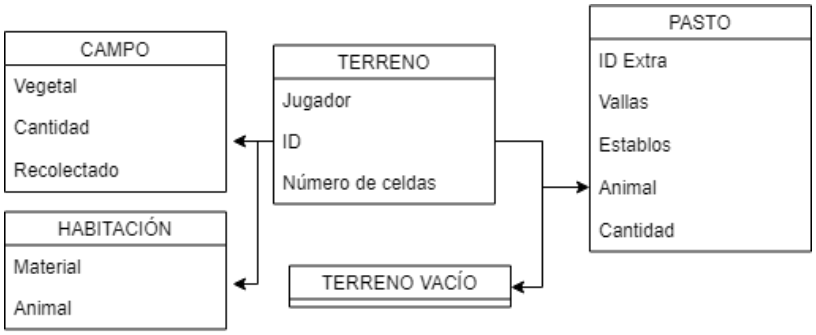

Figura 4: Jerarquía de la clase `TERRENO`.

Para construir, alimentar a la familia y mejorar la puntuación final necesitamos `RECURSOS`. Estos recursos pueden ser de `Tipo: Animal, Vegetal, Material o Comida`. Están asociados a `JUGADORES` de la misma manera que los conceptos definidos previamente: mediante un atributo homónimo de tipo `INTEGER`. Los recursos deben de tener un atributo que nos permitan comprobar si han sido `Puntuados` o no en la última fase del juego, además de un atributo que compruebe si hay `Crías` o no en caso de ser tipo animal (`TRUE/FALSE`).

Para cada tipo de recurso diferente hemos creado una subclase, siendo éstas: `MADERA`, `ADOBE`, `PIEDRA`, `JUNCO`, `HORTALIZA`, `CEREAL`, `OVEJA`, `VACA` y `JABALÍ`.

A pesar de que en una idea original estas subclases estaban diferenciadas jerárquicamente por tipo y después elemento, se ha tomado la decisión de esta jerarquía reducida para simplificar la implementación en CLIPS.

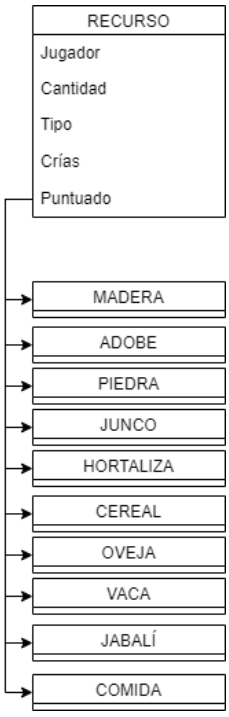

Figura 5: Jerarquía de la clase `RECURSO`.

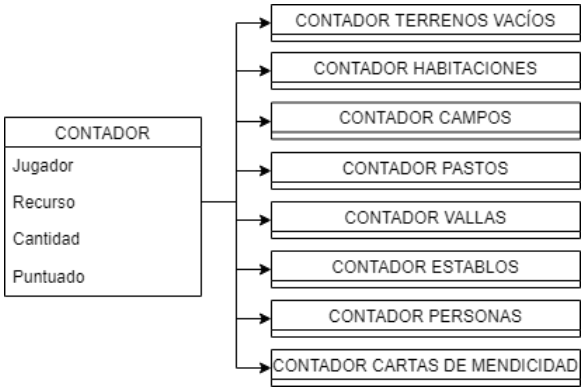

Figura 6: Jerarquía de la clase `CONTADOR`.

El último concepto que necesitamos para tener conceptualizado todo el juego *Agricola* es `CONTADOR`, muy útil para llevar la cuenta de los elementos de cada tipo que tiene cada jugador. Para ello, necesitamos por tanto el atributo `Jugador`, que señala a qué `JUGADOR` pertenece el contador; un atributo `Recurso`, que indica el tipo de recurso que cuenta; `Cantidad`, con la cuenta en sí; y un atributo `Puntuado` con valores `TRUE/FALSE` para contabilizar los puntos al final.

Además, creamos subclases para cada tipo de contador diferente: `CONTADOR-TERRENOS-VACÍOS`, `CONTADOR-HABITACIONES`, `CONTADOR-CAMPOS`, `CONTADOR-PASTOS`, `CONTADOR-VALLAS`, `CONTADOR-ESTABLOS`, `CONTADOR-PERSONAS`, `CONTADOR-CARTAS-DE-MENDICIDAD`. En un principio no existía una jerarquía en esta parte de la ontología, pero no vimos adecuado que cada contador fuera un atributo.

A continuación, presentamos dos diagramas de clases que representan todos los tipos de relaciones (no jerárquicas) que podemos encontrar en nuestra conceptualización de *Agricola*. Resumen asimismo de manera concisa muchas de las relaciones que se pueden encontrar en esta sección y en las que se profundizará una vez expliquemos las reglas.

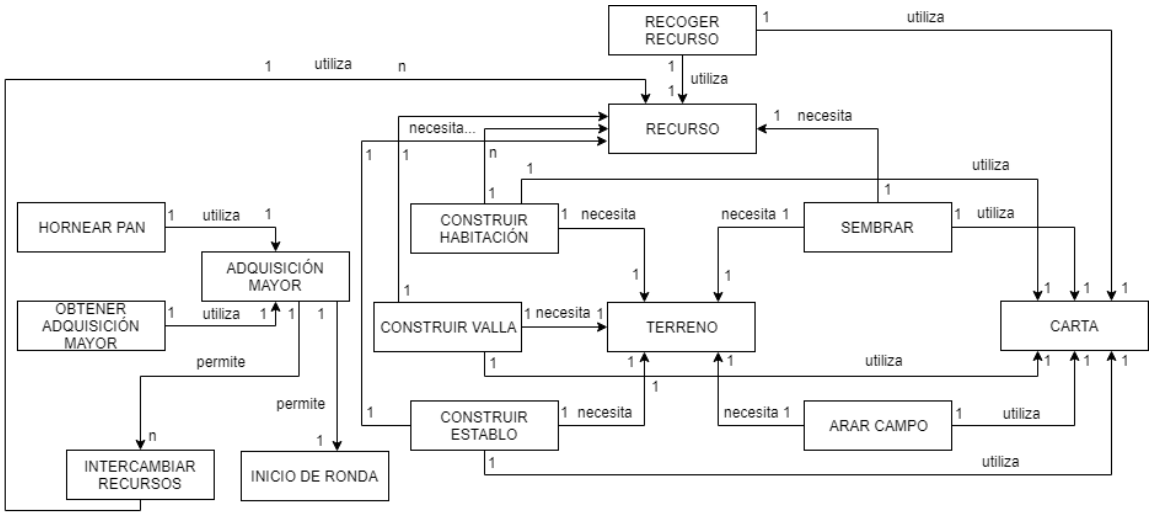

Figura 7: Relaciones no jerárquicas de la clase `ACCIÓN`.

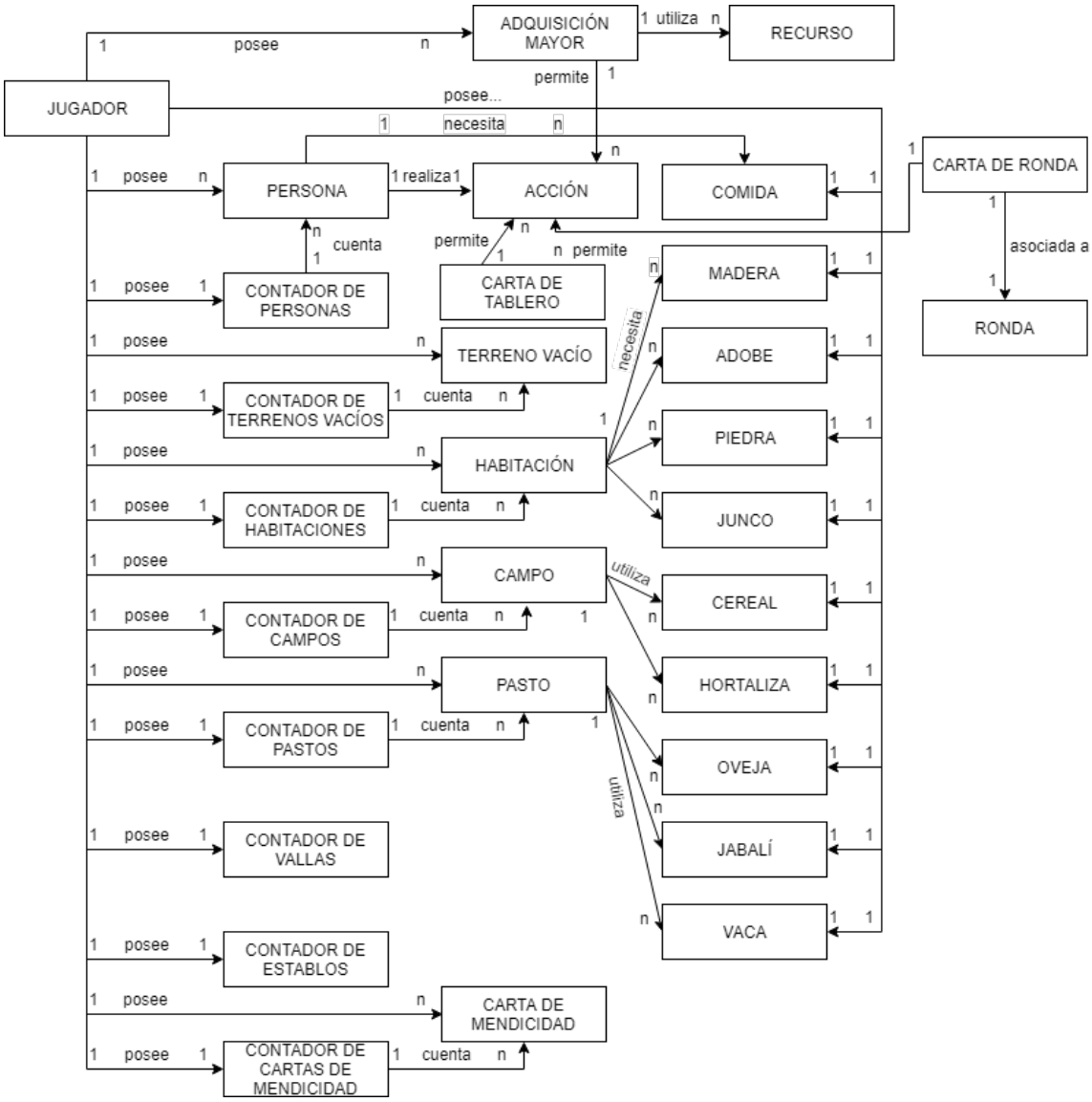

Figura 8: Relaciones no jerárquicas de la ontología.

2.2. Reglas

A la hora de definir las diferentes reglas que iba a tener nuestro Sistema de Producción, decidimos **orientar su implementación a lo que se podía hacer en cada una de las fases del juego**. Así, cada una de las fases tiene su conjunto de reglas específico, a excepción de unas pocas de carácter general.

Es importante indicar que esta implementación está pensada para dos jugadores y que utiliza la versión familiar de *Agricola*. Dicha versión supone una gran simplificación del juego con respecto a la versión avanzada. Sin embargo, y debido a que no se pueden usar funciones aparte de *printout* y *dribble-on* ni otras funcionalidades más avanzadas que ofrece CLIPS, existen otras **restricciones extra**. Algunas de estas restricciones están indicadas en el enunciado de la práctica y otras las hemos incluido nosotras. Estas son:

- **No se incluyen ni las adquisiciones menores ni los oficios**. Las adquisiciones menores son exclusivas de la versión avanzada y, en el caso de los oficios, esto implica que la carta de tablero "Aprender un oficio" no esté incluida.
- **Los pastos sólo pueden estar constituidos de una o dos celdas**, por lo que cualquier otra combinación queda excluida. Por otro lado, tampoco se tiene en cuenta la contigüidad del vallado, por lo que no importa en qué posición se coloquen las vallas. Así, sólo consideramos el número de vallas que puede tener cada pasto, teniendo los pastos simples hasta 4 y los dobles hasta 6.
- **Los establos sólo se pueden construir en pastos ya vallados**. Por tanto, siempre comprobamos que el pasto donde el jugador quiere construir el establo tenga el máximo número de vallas permitidas según su número de celdas.
- **No hemos considerado ninguna restricción de ortogonalidad** para los terrenos que se construyen en el tablero de granja. Así, se podrá construir una nueva habitación, campo o pasto en el terreno o terrenos vacíos que se desee.
- **Para aquellas acciones que son *conjuntas***, es decir, las que indican en su carta correspondiente <acción 1> y <acción 2>; **sólo se comprueba que el jugador pueda realizar la primera de ellas**. Por tanto, es posible encontrar en las trazas casos en los que el jugador ha realizado sólo la primera de las dos. Además, estas acciones se realizan siempre en el orden indicado por la carta.
- **Todas aquellas acciones que contengan la cláusula "y/o" serán tratadas siempre como acciones independientes**. Así, sólo aquellas que contengan la cláusula "y" se ejecutarán de manera *conjunta*.
- En las instrucciones del juego se indica que las **cartas de adquisición mayor** se pueden jugar en cualquier momento de la partida siempre que sea el turno del jugador. Sin embargo, nosotras **hemos limitado su uso a tres situaciones concretas**:
  - A través de la acción "Hornear pan", que se obtiene con la carta de ronda homónima.
  - Al intercambiar los animales por comida tras ejecutar una acción del tipo "Recoger recurso". Esta acción sólo se lleva a cabo cuando no hay ningún pasto ni habitación disponible para guardar al animal.
  - Cuando hay que alimentar a los miembros de la familia y no se dispone de comida suficiente.
- En el caso de que un jugador esté recogiendo recursos animales a través de una **acción del tipo "Recoger recurso"**, se quede sin espacio disponible para los animales restantes (tanto en habitaciones como en pastos) y tampoco los pueda intercambiar por comida por no poseer ninguna carta de adquisición; **dejará dichos animales en el tablero para que se puedan recoger en rondas sucesivas**.

2.2.1. Reglas de carácter general

Las reglas de carácter general son aquellas que no están ligadas a ninguna fase del juego:

- `set-game`. Esta regla se ejecuta siempre la primera porque es la que tiene mayor prioridad de todas (1000) y sólo comprueba que no se haya activado antes haciendo uso del hecho ordenado *strategy*. De esta manera, nos aseguramos que el juego está listo para comenzar dejando la estrategia en *random* y activando la escritura de todos los *printout* en el fichero *salida.txt* haciendo uso de la función *dribble-on*.

- **`ultima-ronda`**. Esta regla cambia la fase del juego a `CALCULO-PUNTUACIÓN` una vez que ha llegado a la ronda 15. El juego tiene 14 rondas, por lo que la ronda 15 no existe; pero esta es una manera mucho más sencilla de comenzar la puntuación que comprobando que nos encontramos en la fase `PROCREACIÓN` de la ronda 14. Para evitar empezar el juego de nuevo en una hipotética ronda 15, le pusimos la segunda prioridad más alta: 500.

2.2.2. Principio de ronda

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase `PRINCIPIO-DE-RONDA`. Las reglas son las siguientes:

- **`cambiar-turno-jugador-inicial`**. Esta regla se activa siempre en esta fase, ya que su intención es la de cambiar el turno de los jugadores independientemente de si actualmente tienen el que les corresponde o no. Tomamos esta decisión con el fin de informar siempre mediante *printout* de quién debía comenzar en cada ronda. Para cambiar el turno, sólo es necesario comprobar qué jugador es el jugador inicial, que es quien empieza cada ronda. Para que esta acción no se ejecutara de forma indefinida, creamos el hecho ordenado *cambiado-turno-jugador-inicial <ronda>*; cuyo uso es el mismo que dimos a *strategy* en `set-game`. Es importante resaltar que todos aquellos hechos ordenados que indiquen un estado del juego que sólo dura una ronda, guardarán el número de ronda en que fueron creados para no tener que utilizar reglas específicas para borrarlos al final de cada ronda (ya que en este caso interferirían con el estado de la nueva ronda).

- **`obtener-carta-ronda`**. Esta regla tiene la tarea de elegir una nueva carta de ronda de entre todas las que no han sido destapadas todavía (*jugador 0*) y que pertenezcan al mismo periodo en que se encuentra el juego. Una vez que se activa esta regla con una carta de ronda concreta, modifica su *slot jugador* a 3, para indicar que puede ser usada por ambos jugadores. Finalmente, y para no destapar más de una carta por ronda, se ha creado el hecho ordenado *carta-de-ronda-jugada <ronda>*.

- **`accion-inicio-de-ronda`**. Hay algunas acciones que se deben ejecutar en esta fase siempre y cuando algún jugador las haya conseguido. En la versión del juego que hemos implementado, esto sólo sucede con la adquisición mayor *Pozo*. Debido a esto, la regla está adaptada a los efectos que produce dicha carta, que es conseguir una unidad de comida en los 5 turnos consecutivos después de haber sido obtenida. Por tanto, esta regla no serviría si la funcionalidad del resto de acciones de este tipo difiere. En cuanto a su funcionamiento, siempre que una acción `INICIO-DE-RONDA` tenga su *slot jugador* con el ID de uno de los dos jugadores (1 ó 2) y su contador no haya llegado a 0, sumaremos uno a la *cantidad* del recurso comida del jugador implicado y restaremos uno al *contador*. Para evitar que esta regla se active en más de una ocasión, se creó el *slot recogido* como variable de control.

- **`resetear-persona-recien-nacido`**. Todas las personas que se crean durante la fase `JORNADA-LABORAL` son recién nacidos. Sin embargo, este estado no debe durar más que una ronda, por lo que deben cambiar a adultos una vez que la ronda ha terminado. Este tipo de acciones las llevamos a cabo normalmente en la fase de `REGRESO-AL-HOGAR`, pero este caso es diferente, ya que que una persona sea o no un recién nacido afecta a la fase de `ALIMENTACIÓN`, que viene después que `REGRESO-AL-HOGAR` siempre que haya cosecha. En consecuencia, decidimos cambiar a todos los recién nacidos a adultos en esta fase, haya o no cosecha en la ronda.

- **`cambiar-fase-inicio-de-ronda`**. La tarea de esta regla no es más que cambiar de fase, más concretamente de `PRINCIPIO-DE-RONDA` a `RECOLECCIÓN`. Existe al menos una regla homónima a esta en cada fase del juego para ir cambiando de ronda. Por tanto, sabemos que se podría haber generalizado si creáramos algún tipo de tupla con cada dos fases consecutivas, pero esta implementación nos resulta más sencilla y es la que se ha quedado. Hay que indicar también que este tipo de reglas siempre tienen la prioridad más baja (1), para asegurarnos de que sean las últimas de la ronda en activarse.

2.2.3. Recolección

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase `RECOLECCIÓN`. Las reglas son las siguientes:

- **`reponer-recursos`**. A través de esta regla se reponen todos los recursos de aquellas acciones que sean del tipo `OBTENER-RECURSO` que estén en juego, es decir cuyo valor de *jugador* sea 3. Por tanto, su tarea es la cambiar el *slot total* de las mismas, sumándole la cantidad a añadir por ronda, guardada en el *slot ronda*. Para evitar reponer recursos hasta el infinito y que sólo se haga una vez por cada acción, utilizamos el *slot repuesto*.

- **`cambiar-fase-reposicion`**. Cambia la fase de `PRINCIPIO-DE-RONDA` a `REPOSICIÓN` una vez que todas las acciones que permiten obtener recursos hayan sido repuestas.

2.2.4. Jornada laboral

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase de `JORNADA-LABORAL`. Estas reglas representan en su mayoría las distintas acciones que puede llevar a cabo cada jugador de forma voluntaria durante la partida, por lo que es en esta fase donde se pueden aplicar distintas estrategias para conseguir más puntos. **Hemos aplicado distintas estrategias básicas para que el comportamiento de nuestros jugadores sean lo más coherente posible**. Existen unos mecanismos comunes que se aplican a todas las reglas:

- En todas aquellas **reglas que no tengan carácter recursivo**, se comprueba cuál es el jugador que tiene el turno para modificar las distintas instancias asociadas a él según indique la acción. Además, se comprueba siempre que no esté *ocupada*, es decir, que ningún jugador la haya asignado a alguna de las personas a su cargo anteriormente en la ronda y también que exista al menos una persona del jugador sin acción asignada (`accion = NULL`). Tras esto, se cambia el turno al jugador que no lo tenía, la acción cambia su estado a *ocupada* y se le asigna la acción a la persona elegida. En el caso de que haya más de una instancia con el *nombre* de esa acción y la *ronda* actual, se crea el estado *cambiar-carta-entera-ocupada* para que la regla `cambiar-ocupada` las cambie a *ocupada* también.

- En el caso de las **reglas recursivas**, no se comprueba qué jugador tiene el turno, sino simplemente si existe el estado correspondiente para su activación, que siempre indica con qué *jugador* y *acción* se debe trabajar; además de la *ronda*, para no tener que borrarla posteriormente. Tampoco es necesario comprobar si la acción se encuentra *ocupada* o no ni si existe alguna persona sin acción asignada.

- Las **reglas *conjuntas***, es decir, aquellas que encadenan dos acciones, funcionan de forma similar a las recursivas. En nuestra implementación, siempre se activa primero la primera de las acciones que hay en la carta, es decir, la que tiene ID 1. Así, son estas las que comprueban el turno del jugador, al igual si la acción ya estaba *ocupada* o no y si se puede asignar a alguna persona del jugador; creando el estado *conjunta*, que debe indicar el *nombre* de la acción, así como el *jugador* que la debe realizar y la *ronda* actual. Este hecho no ordenado es lo único que necesitan aquellas acciones *conjuntas* con ID 2 para activarse.

- En cuanto a las **prioridades**, de forma general, aquellas reglas que no tienen carácter recursivo tienen *salience* 100; mientras que las recursivas tienen *salience* 200 para activarse siempre que exista su estado correspondiente; y finalmente las acciones *conjuntas* con ID 2 tienen *salience* 150, para no interferir con las acciones recursivas, pero activarse antes que la siguiente acción del jugador que tiene el turno.

Las reglas son las siguientes:

- `obtener-recurso`. Esta regla trata el comportamiento de todas las acciones del tipo `OBTENER-RECURSO`. Sin embargo, existe una gran diferencia entre aquellos recursos que son animales con aquellos que no; ya que no tienen el mismo tratamiento una vez recogidos (los primeros se deben colocar en la granja del jugador mientras que los segundos solamente es necesario guardarlos en la reserva personal). Por tanto, hemos definido dos casos diferenciados:
  - `obtener-recurso-no-animal`. Este caso contempla todos aquellos recursos que no son de tipo `ANIMAL`. Lo primero que comprueba es que la *carta* se encuentre disponible para ambos jugadores ($jugador = 3$) y, tras esto, suma al recurso correspondiente de la reserva jugador todas las unidades que se habían acumulado. Finalmente, la cantidad del recurso de la acción pasa a 0.
  - `obtener-recurso-animal`. Este caso contempla todos aquellos recursos que son de tipo `ANIMAL`. Tiene una mayor complejidad que el caso anterior, ya que los animales recogidos se pueden guardar en una habitación o un pasto; o convertirlos en comida. Así, hemos necesitado cuatro reglas distintas para definir estas posibilidades: `obtener-recurso-animal-colocar-en-habitacion`, `obtener-recurso-animal-colocar-en-pasto-vacio`, `obtener-recurso-animal-colocar-en-pasto-no-vacio`, `obtener-recurso-animal-cambiar-por-comida`. Hemos necesitado crear dos reglas para los pastos debido a que cada pasto guarda información acerca del tipo de animal que acoge, por lo que esto es un paso obligatorio para aquellos que no contienen ningún animal, pero no para los que ya tienen alguno. Por otro lado, se ha dado una menor prioridad a intercambiar un animal por comida que al resto de reglas, a modo de estrategia, ya que los animales no dan puntos convertidos en comida y esta acción se puede llevar a cabo en la fase de `ALIMENTACIÓN` si es necesario. Así, lo que hacemos para cada uno de los tres casos, además de comprobar que la carta asociada a la acción está disponible para ambos jugadores, es:
    - Para **incluir un animal en una habitación** es necesario que no exista ningún animal en su interior (`animal = NULL`). En caso afirmativo, la habitación pasa a contener el animal y se actualizan los contadores de la acción y del nuevo recurso del jugador.
    - Para **introducir un animal en un pasto** es necesario que el pasto esté totalmente vallado (aplicamos una fórmula que se basa en el número de celdas), que guarde animales del mismo tipo que el que queremos añadir y que todavía haya espacio suficiente. Si se lleva a cabo la acción, se actualizan los contadores de los recursos que quedan en la acción, el número de animales en el pasto y de la cantidad total del animal que tiene el jugador.
    - Para **intercambiar el animal por comida**, necesitamos que el jugador implicado posea alguna carta de adquisición mayor asociada a una acción del tipo `INTERCAMBIAR-RECURSO` que intercambie el tipo de animal elegido por comida. Si esto es así, se resta una unidad del recurso a la acción (hemos comprobado que en caso de los animales siempre se necesita una unidad para obtener comida) y se suma la comida obtenida a la reserva personal del jugador.

Como hemos visto, se va extrayendo el número total de animales que hay disponibles uno a uno. Por tanto, hemos definido cuatro reglas homólogas a las anteriores de carácter recursivo para que está acción se siga ejecutando hasta que no queden animales o el jugador no tenga más espacio disponible en la granja ni adquisiciones mayores que intercambien animales por comida.

- **`hornear-pan`**. Esta acción sólo tiene una carta asociada, que es la de "Sembrar y/o hornear pan". Se trata de una acción un tanto especial porque necesita dos cartas diferentes para poder llevarse a cabo, algo que no vemos en ninguna otra parte de la implementación. Así, para la primera carta (la mencionada anteriormente) sólo debemos comprobar que se encuentra disponible para ambos jugadores y en el caso de la segunda, que el jugador implicado posea alguna adquisición mayor que permita hornear pan, es decir `hornear-pan` $\neq$ `0`. Tras esto, se resta el número de recursos necesarios para hornear pan al jugador y se le añade a su reserva personal la comida obtenida.

- **`arar-campo`**. Esta acción solamente puede realizarse cuando el jugador tenga un terreno vacío y exista una *carta* que permita arar campos disponible para todos los jugadores. De todas las cartas que hemos utilizado, solamente existe una variación de esta acción: "Arar 1 campo", así que no es necesario crear reglas recursivas. Sus efectos consisten en la transformación de un terreno vacío en un campo, lo que afecta también a los contadores de ambos.

- **`sembrar-campo`**. Para esta acción necesitamos cuatro reglas diferentes, ya que existen dos variaciones posibles para sembrar un terreno (podemos sembrar cereales u hortalizas) y además permite la recursividad, es decir, sembrar más de un campo. Las dos primeras reglas mencionadas podrían haber sido sólo una si el número de unidades de vegetal que debemos depositar en el campo fuera el mismo, pero al no ser así hemos tenido que partirlas en dos. De igual manera que el resto de acciones, necesitamos una carta que permita la acción disponible para todos los jugadores y, en este caso en particular, que el jugador disponga de al menos un campo sin ningún vegetal sembrado (`vegetal = NULL`), al igual que al menos una unidad de cereal u hortaliza. En cuanto a las reglas recursivas, estas han sido implementadas utilizando un poco de estrategia, ya que la acción permite sembrar un número arbitrario de campos que varía desde uno a un máximo que depende del número de cereales y hortalizas de la reserva personal y del número de campos sin sembrar. Esta estrategia consiste en maximizar el número de campos sembrados. Para que sea posible realizar esta recursividad, hemos creado un hecho ordenado llamado `sembrando` con el *id* del jugador, el *nombre* de la carta que estamos utilizando en la regla y el número de la *ronda* en la que sucede esta acción. Este hecho permanecerá activo hasta que no queden más vegetales en la reserva personal del jugador (se pueden plantar cereales y hortalizas indistintamente durante esta acción), momento en el que se activará la regla `quitar-estado-sembrando-no-vegetal` o cuando no queden más campos disponibles, donde se llama a `quitar-estado-sembrando-no-campo`. Estas reglas se crearon porque si el jugador obtenía nuevos campos o vegetales en la misma ronda que había sembrado, al no quitar el estado que activa estas reglas recursivas volvía a sembrar de nuevo.

- **`construir-valla`**. La finalidad de esta acción no es más que construir una valla en un terreno vacío y convertirlo en pasto o en algún pasto ya existente. El juego permite construir tantas vallas como el jugador desee siempre dentro de sus posibilidades. Para que el jugador no se ponga a construir vallas sin ningún criterio, hemos añadido una estrategia. Dicha estrategia consiste en no construir vallas en un terreno vacío hasta que no se hayan terminado de vallar todos los pastos existentes y construir el máximo número de vallas posibles, hasta que el jugador se quede sin madera o sin vallas disponibles (sólo puede utilizar 15). Incluir una estrategia que implica cierta complejidad, además de que en uno de los casos construir vallas se trata de una acción *conjunta*, ha disparado el número de reglas de esta acción hasta 9. Sin embargo, solamente existen tres acciones básicas:

- **`construir-valla-nuevo-pasto-simple`** y **`construir-valla-nuevo-pasto-doble`**. Al crear un nuevo pasto, podemos elegir si este se trata de un pasto simple (formado por una celda) o de un pasto doble (formado por dos celdas). Al tener una definición ligeramente distinta (los pastos dobles deben indicar cuál es su segunda celda en el *slot* *id-extra*), no hay más remedio que crear dos reglas diferenciadas. Para poder ejecutar estas reglas, se debe contar con una carta que permita la acción de construir vallas disponible para todos los jugadores, al igual que uno (o dos terrenos vacíos) donde colocaremos los pastos, al menos una unidad de madera en la reserva personal, no sobrepasar el número máximo de vallas por jugador y la condición de que no exista ningún pasto (en otra posición a donde queremos colocar nuestro nuevo pasto) a menos que no esté completamente vallado. Cumpliendo con estas condiciones, cualquiera de estas dos reglas se puede activar, lo que crearía un nuevo pasto en la o las celdas seleccionadas con una valla y modificaría los contadores de terrenos vacíos y pastos, así como la cantidad de madera que posee el jugador. Hay que indicar que al no permitir construir establos a menos que no sea en pastos ya vallados y que los animales sólo se pueden incluir en un pasto si este está completamente vallado o posee algún establo, todos los pastos se crearán al construir en ellos la primera valla.
- **`construir-valla-pasto-existente`**. Esta regla construye una valla en algún pasto ya creado con anterioridad. Al no permitir construir un nuevo pasto hasta que los actuales no hayan sido totalmente vallados; sólo es necesario incluir en esta regla las condiciones de madera necesaria, el número de vallas máximo por jugador y la cantidad máxima de vallas según el número de celdas del pasto. Así, el comportamiento de los jugadores será siempre el mismo: crearán un nuevo pasto simple o doble que seguirán vallando hasta que se queden sin madera o vallas y en caso de que suceda lo primero, crearán otro nuevo pasto al que le darán el mismo tratamiento que el anterior.

Tres de las reglas restantes existen para poder hacer que esta acción sea recursiva, a través del hecho ordenado `vallando`, que incluye el *id* del jugador, la *carta* de donde se obtuvo la acción y la *ronda* actual. Las otras tres reglas son necesarias para la acción *conjunta* "Reformar y después construir vallas". Para estos casos, la acción *conjunta* pasa a ser recursiva y se comporta como hemos visto en los puntos anteriores. Por último, y para evitar que el jugador vuelva a construir vallas porque ha obtenido más madera durante la ronda, se ha creado la regla `quitar-estado-vallando-no-madera`, que se activa, como su propio nombre indica, cuando el jugador se queda sin madera. No es necesario eliminar este hecho ordenado tras llegar al número máximo de vallas, ya que las propias reglas impedirán una nueva activación con sus restricciones. Esta es una constante que veremos en las próximas reglas que necesiten de esta intervención: sólo tenemos en cuenta los recursos necesarios para llevarlas a cabo.

- **`construir-establo`**. Esta acción consiste en construir un número indeterminado de establos, entre uno y los que permitan los pastos totalmente vallados que posea el jugador. En este caso, hemos introducido la estrategia de maximizar el número de establos a construir. Para ello, sólo necesitamos dos reglas: una que construya el primer establo y otra que lo haga de forma recursiva, al igual que sucedía con `sembrar-campo`. Esta acción transforma dos unidades de madera de la reserva personal del jugador en un establo, lo que permite aumentar la capacidad máxima de animales por pasto. Además de la condición del número de recursos necesarios, la regla comprueba que no se ha llegado al máximo número de establos por jugador (4), que el pasto donde queremos construir el establo está totalmente vallado y que no se ha alcanzado el número máximo de establos por pasto (uno por celda). Tras esto, restaremos las unidades de madera correspondiente al jugador y actualizaremos el número de establos del pasto y el contador de establos del jugador. La primera vez que se active esta acción se creará el hecho ordenado `construyendo-establos`, con el *id* del jugador asociado y el *nombre* de la carta asociada a la acción, además del número de la *ronda* en la que se activa. Este estado no se eliminará de la base de hechos hasta que el jugador se quede sin madera suficiente para construir más establos,

momento en el que se ejecutará la regla `quitar-estado-construyendo-establos-no-madera`. Con esto evitaremos que el jugador vuelva a construir establos después de obtener madera.

- **`construir-habitación`**. Durante la partida, los jugadores pueden cambiar un `TERRENO-VACÍO` por una `HABITACIÓN` del mismo material que las que ya se encuentran situadas en la granja del jugador. Al igual que sucedía con los establos, se pueden construir tantas habitaciones como los recursos del jugador y el máximo de habitaciones (5) le permitan. Por tanto, esta acción podrá ser utilizada siempre que el jugador tenga entre sus recursos personales al menos dos unidades de `JUNCO` y cinco unidades del material que será la vivienda, ya sea `MADERA`, `ADOBE` o `PIEDRA`, y que disponga de la *carta* que les permite realizar esta acción. Tras esto, se deben actualizar los recursos y contadores correspondientes y crear el hecho ordenado de `construyendo-habitaciones`. Si el jugador no tiene suficiente madera, adobe o piedra para construir para seguir con esta tarea, se activará `quitar-estado-construyendo-habitaciones -no-madera-adobe-piedra` y en caso de que sean los juncos los que escaseen, se llamará a `quitar-estado-construyendo-habitaciones -no-juncos`. Con esto evitaremos que el jugador pueda construir nuevas habitaciones después de haber conseguido nuevos recursos.

- **`reformar-habitación`**. Las reglas que conforman reformar habitación necesitan hacerse de manera conjunta, es decir, necesitamos los recursos suficientes para mejorar todas las habitaciones a la vez. Al igual que hemos explicado en reglas anteriores, necesitamos una primera regla que determine que podemos reformar la vivienda y cree un hecho ordenado, en este caso `reformando-habitaciones`, que permita identificar cuándo es necesario acceder a las reglas recursivas con el mismo nombre. A diferencia de otros hechos ordenados utilizados para la recursividad, en este caso debemos indicar cuál es el material que se va a utilizar para reformar, ya que tendremos habitaciones con el material antiguo y el nuevo coexistiendo durante el proceso recursivo y no hay ninguna forma de saber cuál de los dos es el correcto. Además, las habitaciones se pueden mejorar de madera a adobe y de adobe a piedra. Al no haber incluido ningún hecho o clase que relacione a ambas en la base de hechos, hemos creado dos reglas con el mismo funcionamiento pero transiciones diferentes: `reformar-habitacion-madera-a-adobe` y `reformar-habitacion-adobe-a-piedra`. Así, cada vez que reformamos una habitación, modificamos los contadores de los recursos utilizados y cambiamos el material. Por último, debemos indicar que todas las acciones que implican reformar habitaciones son conjuntas con ID 1, por lo que no es preciso hacer ninguna distinción entre acciones *conjuntas* y aquellas que no lo son. Por tanto, todas las reglas, a excepción de las recursivas, crean el estado *conjunta*.

- **`ampliar-familia`**. Esta acción puede suceder de dos maneras: antes del quinto periodo se necesita disponer previamente de una habitación libre, por lo que solamente se puede ejecutar la regla `ampliar-familia-con-habitación`, pero después, cuando entra en juego la *carta* que permite ampliar la familia sin esa necesidad, se puede utilizar `ampliar-familia-sin-habitación`. En ninguno de estos casos necesitamos la recursividad, ya que esta acción consiste en crear una nueva persona, no varias a la vez. Así, ambas reglas crean una instancia `PERSONA` nueva con valor de `recién-nacido` en `TRUE` e incrementan el `CONTADOR-PERSONAS` del jugador correspondiente. La diferencia entre las dos se encuentra en que cuando se pide una habitación, hay que comprobar que existen más habitaciones que personas; pero en ambos casos no se debe superar el límite de personas por jugador (5).

- **`obtener-adquisicion-mayor`**. Esta acción permite comprar una carta de adquisición mayor a un jugador siempre que este disponga de los recursos necesarios para ello. Podemos dividir las adquisiciones mayores en dos grandes grupos: aquellas que sólo necesitan un tipo de recurso y aquellas que necesitan dos (como pudimos ver en nuestra ontología). Esta división es necesaria porque CLIPS no puede manejar ambos casos a la vez en una misma regla, por necesitar un número distinto de instancias de `RECURSO`. El caso de la primera se encuentra representado por `obtener-adquisicion-mayor-1-recurso`, que pide que la adquisición mayor a conseguir no haya sido comprada por nadie (*jugador* = 0) y que el jugador

tenga la cantidad necesaria del recurso. Esto se traduce en otorgar la titularidad de la carta al jugador que realizó la acción y en restarle la cantidad del recurso que ha gastado para obtenerla. Por otro lado, las cartas con dos recursos presentan una particularidad, y es que no en todas piden recursos en exclusiva, sino también la carta "Hogar". Debido a esto, hemos tenido que dividir esta regla en dos: `obtener-adquisicion-mayor-2-recursos-no-hogar` y `obtener-adquisicion-mayor-2-recursos-hogar`. El caso de la primera es sencillo, ya que sólo hay que hacer lo mismo que hemos hecho con las cartas que pedían sólo un recurso, pero con dos recursos. Sin embargo, en aquellos casos en los que se pide un hogar, el jugador tiene dos opciones: entregar dicha carta o entregar los recursos. Así, y como parte de la estrategia, consideramos que es mejor mantener la carta "Hogar", por lo que hemos dado mayor prioridad a intercambiar la adquisición mayor por recursos. Por tanto, el segundo caso funciona igual que las adquisiciones mayores que piden un único recurso (con la particularidad de que comprueba que el primer recurso es `HOGAR`) y el primero se traduce en quitar la carta "Hogar" al jugador (*jugador* = 0) y en entregarle la que haya comprado. Finalmente, hay que destacar que esta acción puede ser parte de una acción *conjunta* con ID 2, por lo que todas las reglas mencionadas anteriormente tienen su versión homóloga para este caso. A diferencia de `construir-vallas`, en este caso debemos quitar el estado de *conjunta* en cuanto se realiza la acción, para evitar que el jugador compre más de una adquisición mayor.

- **`cambiar-jugador-inicial`**. Esta acción puede ser utilizada por cualquier jugador, aunque el que la use ya sea el jugador inicial, para asegurarse de que mantiene su estatus en la siguiente ronda. Es una regla sencilla en la que se intercambia el valor del atributo de *jugador inicial* entre ambos jugadores, teniendo en cuenta cuál de los dos ha llamado a la regla. Toma efecto a partir de la siguiente ronda, cuando ejecutamos `cambiar-turno-jugador-inicial`.

- **`cambiar-turno-familias-desbalanceadas`**. En caso de que un jugador posea más personas que otro, llegará un momento en el que uno de los dos haya terminado de realizar sus posibles acciones y al otro aún le queden por hacer. Por eso necesitamos esta acción, para cambiar el turno del jugador que aún tiene posibilidades de continuar. Tras estudiar distintas combinaciones de número de personas entre ambos jugadores, vimos que la condición necesaria para devolver el turno al otro jugador era que tuviera más personas a su cargo. Esta regla tiene la menor prioridad de todas las que conforman la `JORNADA-LABORAL`, a excepción de la regla de cambio de fase; con el objetivo de que sólo pueda activarse cuando el jugador que tiene el turno no tenga ninguna persona sin acción asignada (`accion` $\neq$ `NULL`).

- **`cambiar-fase-jornada-laboral`**. Cambia la fase de `JORNADA-LABORAL` a `REGRESO-AL-HOGAR` una vez que cada persona perteneciente a cada jugador haya ejecutado una acción.

2.2.5. Regreso al hogar

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase `REGRESO-AL-HOGAR`. En esta fase solamente reseteamos las instancias necesarias para activar más reglas en la ronda siguiente, mediante la modificación de ciertos atributos. Las reglas utilizadas son las siguientes:

- **`resetear-recolectado`**. Cambia todos los campos que han sido recolectados (`recolectado = TRUE`) a `FALSE`, para que poder recolectar vegetales de nuevo en rondas sucesivas.

- **`resetear-persona-adulto`**. Esta regla cambia la acción asignada a todas las personas adultas a `NULL` y su *accion-id* a 0, para que se les pueda asignar nuevas acciones en la ronda siguiente.

- **`resetear-accion`**. Pone todas las acciones con valor `ocupada = FALSE` a `TRUE`, para que los jugadores puedan realizar acciones en la ronda que viene.

- **resetear-inicio-de-ronda**. Cambia el atributo recogido de las acciones INICIO-DE-RONDA de TRUE a FALSE. Así, en caso de haber utilizado la acción, se podrá volver a activar en la ronda siguiente.
- **resetear-recoger-recurso**. Cambia a FALSE todos los recursos que han sido repuestos en la ronda actual, para que puedan actualizarse en la siguiente.
- **resetear-recurso**. Establece el valor de las crías de los animales que se han reproducido en la ronda anterior a FALSE.

Al igual que en el resto de fases, tenemos la regla **cambiar-fase-regreso-al-hogar** que se activará cuando hayamos reseteado todos los valores de instancias explicados previamente para pasar a la fase PRINCIPIO-DE-RONDA.

En ciertas ocasiones, cuando la ronda coincide con el fin de un periodo, esta continúa con tres fases más (las pertenecientes a la COSECHA). En este caso, en lugar de activarse la regla mencionada previamente, se activaría **cambiar-fase-regreso-al-hogar-cosecha**, que cambia la fase a RECOLECCIÓN, ya que es esta la que tiene mayor prioridad.

2.2.6. Cosecha - Recolección

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase RECOLECCIÓN. La fase de recolección solamente se centra en recoger los campos sembrados para poder disponer de recursos suficientes en la reserva personal en caso de que los jugadores los necesiten para alimentar a sus familias. Hay que indicar que una vez entramos en las fases pertenecientes a la cosecha, se elimina el sistema de turnos, ya que el orden de las acciones entre ambos jugadores se vuelve irrelevante. Las reglas son las siguientes:

- **recolectar**. Recoge una unidad de cada campo que posea el jugador, tanto CEREALES como HORTALIZAS, y la almacena en la reserva personal.
- **resetear-campo**. Cambia todos los campos que se han quedado sin unidades del vegetal que estaban cultivando a vegetal = NULL, ya que esta es una condición necesaria para poder seguir sembrando en ellos. Incluimos esta regla aquí y no en REGRESO-AL-HOGAR porque afecta directamente a la fase de JORNADA-LABORAL, que viene antes.
- **cambiar-fase-recolección**. Una vez recolectados y reseteados todos los campos (si fuera necesario), cambiamos a la fase ALIMENTACIÓN.

2.2.7. Cosecha - Alimentación

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase ALIMENTACIÓN. El objetivo de todas las reglas de esta fase es alimentar a la familia con diferentes recursos. En esta fase hemos forzado cierta estrategia para conseguir la mayor cantidad de comida en caso de no tenerla y evitando las cartas de mendicidad a toda costa. Las reglas que componen esta fase son las siguientes:

- **Prioridad 1: alimentar a las personas con la comida que se posee**. Estas dos reglas que explicamos a continuación son las que tienen mayor prioridad. De este modo, el jugador sólo buscará maneras de obtener comida cuando no pueda alimentar ni a adultos ni a recién nacidos.
  - **alimentar-recién-nacido**. Las reglas para alimentar personas no han podido diseñarse juntas, ya que los recién nacidos necesitan de menos unidades de COMIDA para ser alimentados que los adultos. En este primer caso, solamente necesitamos una unidad.
  - **alimentar-adulto**. Para los adultos realizamos la misma acción. Sin embargo, necesitamos dos unidades de COMIDA en lugar de una.

- **Prioridad 2: usar adquisiciones mayores para intercambiar recursos por comida**. Como sabemos que que cualquier adquisición mayor da mucha más comida que usar un vegetal, esta es la primera opción que ofrecemos a los jugadores cuando no tienen comida suficiente.
  - **`intercambiar-recurso-no-animal-comida`**. Si llegamos a la situación de que un jugador no disponga de suficientes unidades de `COMIDA` para alimentar a su familia, existe la posibilidad de cambiar recursos por comidas mediante el uso de cartas de adquisición mayor. Estas cartas pueden cambiar tanto recursos no animales (como es el caso de esta regla) como recursos animales, explicado en la regla siguiente. Esta división en cuanto al tipo de recurso es vital, ya que cuando usamos un recurso no animal, simplemente lo extraemos de la reserva personal del jugador, mientras que si es animal, debemos retirarlo de la habitación o pasto donde se encuentre.
  - **`intercambiar-recurso-animal-comida-quitar-de-habitación`**. Esta regla se activará si el jugador dispone de cartas de adquisición mayor que permitan cambiar animales por `COMIDA` y tenga a su vez un animal de compañía en una de sus habitaciones. Esta acción cambia un animal por unidades de `COMIDA`, que dependen de la carta utilizada. Es necesaria hacer una distinción entre pasto y habitación, ya que, al poder incluir más de un animal en un pasto, necesitamos indicar los animales que quedan, mientras que en la habitación sólo hay que decir si hay animal o no.
  - **`intercambiar-recurso-animal-comida-quitar-de-pasto`**. De igual manera que la regla anterior, se usa una carta de adquisición mayor que posea el jugador para cambiar un animal (que esta vez que se encuentra en un pasto) por diferentes unidades de `COMIDA`, dependiendo de la carta utilizada.
  - **`resetear-pasto`**. En caso de que el pasto se quede vacío tras quitar un animal, debemos quitarle el tipo de animales que guarda, para que poder guardar cualquier otro en rondas sucesivas. Si no hiciéramos esto, obligaríamos al jugador a guardar el mismo tipo de animales en los pastos, a pesar de que ya no haya animales de dicho tipo.
- **Prioridad 3: Cambiar vegetales por comida**. Si el jugador no tiene ninguna carta de adquisición mayor o se ha quedado sin los recursos que necesita para intercambiar comida con las adquisiciones mayores que posee, entonces no tendrá más remedio que usar los vegetales de su reserva personal.
  - **`cambiar-vegetal-comida`**. Esta acción convierte una unidad de cereal o de hortaliza en una de `COMIDA`. Al principio, esta regla estaba dividida en dos, pero al establecer el *slot tipo* en `RECURSO`, las hemos podido juntar en una comprobando que el recurso es de tipo `VEGETAL`.
- **Prioridad 4: Mendigar**. Mendigar es siempre el último recurso al que debe acudir cualquier jugador, ya que empeora mucho la puntuación. Por tanto, es la regla que tiene la menor prioridad a excepción de la cambia de fase.
  - **`mendigar`**. Si un jugador no dispone de suficientes recursos para alimentar a su familia, necesitará de tantas cartas de mendicidad por personas sin alimentar. En esta regla se suma una carta de mendicidad por persona sin alimentar y cambia su estado a `alimentado = TRUE`.
- **`cambiar-fase-alimentación`**. Una vez que todas las personas han sido alimentadas, pasamos a la siguiente fase de la cosecha, `PROCREACIÓN`.

Como podemos observar, el procedimiento de esta fase sigue la estrategia de alimentar a todas las personas que se pueda hasta que el jugador no tenga comida suficiente, que es cuando intercambiará recursos por comida. En cuanto el jugador tenga la comida necesaria para alimentar a al menos a una persona, dejará de intercambiar recursos por comida. Así, el habrá un momento en que se intercambien acciones de alimentar y de obtener comida. En caso de no tener suficiente con esto, todas las acciones finales consistirán en obtener cartas de mendicidad.

2.2.8. Cosecha - Procreación

Todas las reglas que explicaremos a continuación comprueban siempre que el estado del juego se encuentra en la fase `PROCREACIÓN`. En esta fase, los animales que tengan al menos dos unidades entre todos los terrenos de un jugador, podrán tener una cría.

- **`crear-cria`**. Para cada recurso de tipo animal que tenga al menos dos unidades cambiamos el atributo `crias` a `TRUE`. Esta regla es la que tiene mayor prioridad de todas, para saber de qué animales tenemos que repartir las crías, antes de ponernos a ello. El resto de reglas de esta fase, menos la que permite cambiar de fase, tienen una prioridad intermedia (90).
- **`repartir-cria-habitacion`**. Si se posee una cría de animal, se comprueba si hay espacio disponible en una de las habitaciones, para establecer la nueva cría como animal de compañía. Hemos tenido que dividir la regla de `repartir` en habitación y pasto, ya que en la primera sólo hay que indicar si hay un animal o no y de qué tipo, mientras que en la segunda hay que indicar el tipo de animal y la cantidad.
- **`repartir-cria-pasto-vacio`**. De mismo modo, también puede asociarse esa cría a un pasto. Esta regla es necesaria separarla de la siguiente, ya que puede que el pasto no se disponga de un animal hasta el momento o puede que ya existan animales pero no se haya alcanzado la capacidad máxima. Si ocurre lo primero, necesitamos asociar el tipo de cría al pasto.
- **`repartir-cria-pasto-no-vacio`**. En caso contrario, simplemente comprobamos que la cría que estamos introduciendo en un pasto que ya posee animales coincida en tipo.
- **`cambiar-fase-procreacion`**. Por último, cambiamos a la fase `PRINCIPIO-DE-RONDA` cuando no es posible repartir las crías que faltan o ya han sido todas repartidas.

2.2.9. Cálculo de la puntuación

Solo podemos acceder a esta fase en la última ronda del juego (nuestra hipotética ronda 15). Las reglas que la componen solamente suman los puntos que va ganando (o perdiendo) el jugador por cada recurso, terreno o carta considerada, actualizando por pantalla el contador de puntos totales. Las reglas que componen esta fase son:

- **`añadir-vegetales-en-campo-a-reserva`**. En primer lugar, debemos tener en cuenta los recursos que están aún sembrados, puesto que no se han contabilizado todavía en la reserva personal del jugador. Por eso es necesaria esta regla que se activa al principio de la fase, ya que es la que posee mayor *salience*. La única función de esta regla es sumar todos los vegetales disponibles en un terreno `CAMPO` al contador de cereales u hortalizas del jugador, según corresponda. Así, la regla se ejecuta de forma recursiva con cada campo que tiene una cantidad de recursos distinta a 0.
- **`sumar-puntos-recurso`**. Esta regla tiene la función de sumar los puntos que otorga cada `RECURSO` que se tenga en cuenta para la puntuación. Si un recurso no ha sido contado, su atributo `puntuado` aún estará con valor `FALSE`. (Esta es una constante que se utiliza para todas y cada una de las reglas de esta fase.) Para comprobar la equivalencia de puntos que el jugador obtiene por recurso y cantidad, sólo es necesario mirar la instancia correspondiente de `TABLA-PUNTUACIÓN`. Una vez sumados los puntos al contador del jugador, podemos establecer ese recurso como `puntuado = TRUE`. Utilizar esta tabla para los recursos es arriesgado, ya que debemos acotar el número de unidades máximas que tendremos en cuenta para la puntuación. Esto no es complicado en el caso de los animales, ya que con el límite de vallas sólo se pueden tener hasta 16 del mismo tipo. Sin embargo, en el caso de los vegetales no hay límite, así que tras varias pruebas comprobamos que poner ese límite en 30 era suficiente.

- **`sumar-puntos-contador`**. Existen ciertos contadores como el número de terrenos vacíos, el número de campos, pastos, establos... que pueden ser accesibles de la misma manera. Esto se debe a la inclusión del atributo `recurso` dentro de los contadores, gracias al cual podemos obtener la información de la `TABLA-PUNTUACIÓN`. Por tanto, solo necesitamos una única regla que sea capaz de activarse para cada contador y sume los puntos asociados. En estos casos, no hay problema con que la tabla sea acotada, ya que los terrenos pueden ser como máximo 15, los establos 4 y las personas 5.
- **`sumar-puntos-habitaciones`**. No hemos podido incluir este caso en el anterior, porque aparte del número de habitaciones se debe comprobar de qué material están construidas. Así, decidimos crear una regla específica para este caso. Sin embargo, como cada tipo de habitación devuelve una cantidad de puntos diferente, hemos tenido que crear una regla para cada material. En el caso de las habitaciones de adobe, el jugador obtiene un punto por habitación a su contador total. Si el jugador tiene habitaciones de piedra, entonces obtendrá dos puntos por cada una de las habitaciones en su poder.
- **`sumar-puntos-cartas-adquisicion-mayor`**. Para sumar estos puntos necesitamos cuatro reglas diferentes, ya que dependiendo de la carta se pueden sumar 3, 2, 1 o ningún punto extra. Todas estas reglas funcionan de la misma manera, es decir, comprueban que el jugador posee la carta y comprueba, cuando sea necesario, si tiene los recursos suficientes para conseguir los puntos extra; tras lo que añade los puntos por tener la carta. Cada una de estas reglas tiene prioridades distintas para que se activen las reglas por orden descendiente de puntos extra. Con esto nos aseguramos que, en caso de poder recibir puntos extra, el jugador siempre consigue el máximo posible (la carta no se volverá a evaluar de nuevo gracias al atributo `puntuada`).
- **`terminar-ejecucion`**. Una vez ejecutadas todas las reglas anteriores, podemos ver la puntuación total de ambos jugadores y comprobar cuál de los dos ha ganado la partida (o si ha habido un empate). En caso de no empatar, hemos decidido mostrar pon pantalla también la diferencia de puntos además de los puntos finales de cada jugador. Debemos, además, terminar de escribir el fichero (*dribble-off*). Esta regla es la última en ejecutarse debido a su baja prioridad dentro de la última fase accesible del juego. Por tanto, llamará a *halt* para el programa pare.

3. Manual de usuario

Para poder **probar nuestro Sistema Basado en el Conocimiento en un ordenador con sistema operativo Linux**, se debe abrir la terminal, situarse en la carpeta donde se encuentran los ficheros con extensión .clp de esta entrega[1] y ejecutar el comando `clips`. Una vez que nos encontramos dentro de CLIPS, debemos ejecutar los siguientes comandos:

1. `(load ontologia.clp)`, para cargar nuestra ontología.
2. `(load prueba-X.clp)`, para cargar las instancias de cada una de las cinco pruebas.
3. `(load reglas.clp)`, para cargas las reglas.
4. `(reset)`, para actualizar la base de hechos.
5. `(run)`, para que se ejecuten las reglas.

Tras esto, veremos una serie de mensajes por pantalla que muestran cada una de las acciones que han hecho los jugadores, así como las fases y rondas por la que pasa el juego y los resultados finales. Si queremos recuperar todos los mensajes de la ejecución, debemos abrir la carpeta donde se encuentran los ficheros .clp (no tiene por qué ser a través de la terminal) y abrir `salida.txt`. En caso de querer ver cualquier otra prueba, sólo hay que volver a ejecutar las reglas 2, 4 y 5 (en ese orden).

[1] Para ello es necesario utilizar el comando `cd`.

4. Pruebas realizadas

Para poder comprobar el correcto funcionamiento del Sistema de Producción, esperamos a tener todas las reglas programadas para ver cómo interactuaban entre ellas. Tras ello, evaluamos su comportamiento hasta comprobar que era exactamente el que deseábamos. De esas pruebas obtuvimos los casos que ahora presentamos en esta sección, que representan los errores más frecuentes o acciones que representan un mayor nivel de complejidad.

En la **primera prueba** realizada, se introdujo en la Base de Hechos todas las instancias necesarias para simular una partida desde el principio. Elegimos tras varias ejecuciones la que había hecho una partida más interesante: uno de los jugadores tenía animales de más de un tipo en sus pastos, se habían comprado cartas de adquisición mayor, uno de los jugadores tenía varias cartas de mendicidad en su poder, se habían construido establos y vallas y reformado algunas habitaciones. Gracias a esta prueba, comprobamos que el flujo del juego era el correcto: las fases sucedían en el orden determinado y había cosecha cuando habíamos indicado. Además, se comprobó que las acciones recursivas como `vallar`, `sembrar` o `repartir animales` funcionaban correctamente, al igual que aquellas que eran conjuntas, como `reformar`. (Estos dos tipos de acciones han sido las más problemáticas mientras corregíamos las reglas.) Finalmente, se ha comprobado también que las prioridades funcionan como se esperaba, haciendo que aquellas acciones que catalogamos como *malas* no se ejecuten nunca y que el orden según nuestra estrategia sea el esperado, muy interesante de analizar en reglas recursivas y conjuntas y en las fases de `ALIMENTACIÓN` y `PROCREACIÓN`.

La **segunda prueba** que hemos realizado consiste en comenzar la partida con todos los campos dobles posibles, simulando que todos están vallados y que todos tienen el número máximo de establos. A continuación, hemos establecido el número de ovejas que se pueden recoger a un número arbitrariamente grande para comprobar que el jugador va a colocar recursivamente todas las ovejas posibles en sus posibles espacios y que dejará en el tablero el resto (ya que no posee ninguna carta de adquisición). En la salida de esta prueba, podemos observar que el Sistema de Producción no tiene ningún problema en colocar todas las ovejas en los respectivos campos y pasa el turno al siguiente jugador cuando llega a su límite de colocación. Además, ningún jugador vuelve a colocar más animales en los pastos una vez que llega al límite, ni tampoco a permitir que estas procreen. Aquí comprobamos además que no se contabilizan los puntos, puesto que hemos recurrido a la clase `TABLA-DE-PUNTUACIÓN` con valores de la puntuación asociada a las cantidades de cada recurso y solamente existen instancias de un número razonable de recursos que puede alcanzar un jugador en una partida normal, como la que se podría dar en la prueba anterior.

Para la **tercera prueba**, se establece el número de maderas de cada jugador a 40 unidades. Con esto vamos a comprobar que en el momento que se active la regla `construir-vallas`, todas serán colocadas recursivamente hasta que se alcance el máximo número de vallas. De igual manera, se activará `construir-habitación` hasta que haya 5 habitaciones de madera por cada jugador, así como pasará con la regla `construir-establo`, que será ejecutada recursivamente hasta generar el número máximo de establos por pasto. Podemos así comprobar que la recursividad de estas acciones funciona correctamente, ya que para cuando estimamos.

Con la **cuarta prueba** hemos comprobado que un jugador reforme tantas habitaciones como sea posible dentro del mismo turno si tiene posibilidad de utilizar la acción correspondiente. Para alcanzar este objetivo hemos introducido en la Base de Hechos tantas habitaciones como terrenos disponibles, así como un número elevado de madera, adobe, piedra y juncos para cada uno de los jugadores. El otro gran interés que tenemos en esta prueba es ver si los jugadores son capaces de ejecutar correctamente acciones conjuntas, siendo la de reformar una acción conjunta en todas sus variantes. Al ver los resultados de esta prueba, podemos observar en la salida cómo cada jugador ha reformado sus habitaciones al máximo, incluyendo incluso animales (de los tres tipos disponibles) en cada una de ellas. Al poseer tantos recursos, nos hemos dado cuenta que se ha disparado la compra

de adquisiciones mayores, ya que se trata de una de las segundas acciones de estas acción conjunta (la otra es construir vallas, lo cual es imposible porque todos los terrenos son habitaciones). Hemos comprobado también que no se repitieran estas acciones en una misma ronda, ya que ha sido un error común mientras probábamos el código, y qué mejor que hacerlo cuando se dispone de tantos recursos que permiten hacer muchas más acciones.

En la **última prueba** hemos comprobado que la acción recursiva sembrar se ejecuta correctamente teniendo el máximo número de campos y un número abundante tanto de `CEREALES` como `HORTALIZAS`. Así, vemos que cada jugador, en cuanto encuentra la oportunidad, siembra todos sus campos en una misma ronda. Por otro lado, en la fase de `RECOLECCIÓN`, hemos comprobado que se recoge una unidad de todos y cada uno de los campos arados, reduciendo correctamente las unidades del recurso. De este modo, cuando el campo tiene 0 unidades de vegetal, vemos que no tiene ningún problema en volverlos a sembrar incluso en la ronda siguiente. Por otro lado, en la fase de `CÁLCULO-PUNTUACIÓN`, se puede observar que todos los recursos que se encuentran son recogidos de sus campos. No obstante, nos volvemos a encontrar el mismo problema que en la prueba 2, ya que al tener muchos más vegetales de los que se pueden conseguir en una partida normal, estos se salen de los límites de `TABLA-PUNTUACIÓN` y no se tienen en cuenta para la puntuación.

Antes de hacer estas pruebas, encontramos muchos problemas con la recursividad de los campos (el jugador volvía a sembrar cuando conseguía más cereales u hortalizas en la misma ronda), además de que en la fase de recolección, sólo recolectaba cada campo en una ocasión. Sin embargo, podemos concluir que tras estas pruebas y otras muchas que vinieron antes, esta solución propuesta se comporta tal y como hemos descrito en el *Manual técnico.*

Test ID	**Propósito**	**Resultados**	**Estado inicial**	**Salida**
1	Comprobar el correcto funcionamiento de una partida al uso desde el principio	Partida completa ejecutada con el comportamiento deseado.	prueba-1.clp	salida-prueba-1.txt
2	Comprobar la correcta funcionalidad de la recursividad al colocar animales en el pasto de jugador	Funcionamiento correcto. No calcula completamente la puntuación total (se deja las ovejas) por salirse de los márgenes acotados en `TABLA-PUNTUACIÓN`.	prueba-2.clp	salida-prueba-2.txt
3	Comprobar la correcta funcionalidad de la recursividad para todas aquellas acciones que dependen del uso de madera	Funcionamiento correcto.	prueba-3.clp	salida-prueba-3.txt
4	Comprobar la correcta funcionalidad de la recursividad y de las acciones conjuntas para el caso concreto de reformar habitaciones	Funcionamiento correcto.	prueba-4.clp	salida-prueba-4.txt
5	Comprobar la correcta funcionalidad de la recursividad para sembrar los campos y sus acciones derivadas	Funcionamiento correcto. No calcula completamente la puntuación total (se deja los cereales y hortalizas) por salirse de los márgenes acotados en `TABLA-PUNTUACIÓN`.	prueba-5.clp	salida-prueba-5.txt

Cuadro 1: Tabla-resumen de las pruebas realizadas.

5. Conclusiones

Algunas de las conclusiones técnicas que hemos obtenido tras trabajar en este proyecto se pueden resumir en los siguientes puntos:

1. **La parte más importante y compleja** de la creación y desarrollo de un sistema de producción **es la identificación y adquisición del conocimiento**. Una vez que tuvimos claras estas partes, la creación de la ontología y las reglas fue cuestión de programar en CLIPS las ideas trabajadas durante la creación de estos documentos. De hecho, hablando desde nuestra experiencia personal, obtuvimos una primera versión bastante buena de la implementación basándonos en exclusiva en el documento de adquisición del conocimiento.

2. **El tiempo de ejecución es mucho menor que el estimado en la fase de identificación del problema**, ya que se tuvo en cuenta lo que tarda una persona en tomar decisiones. Sin embargo, nuestro Sistema de Producción simplemente lleva a cabo la activación de una serie de reglas computacionalmente sencillas. En consecuencia, la ejecución no tarda más que unos segundos.

3. A la hora de implementar las **reglas, ha resultado complicado poder conseguir un nivel de generalización alto**, ya que sólo por pequeñas variaciones dentro de una misma acción se ha tenido que descomponer una idea mucho más general en otras más específicas. Un ejemplo concreto sería el caso de `sembrar`, que requiere dos reglas diferentes por el simple hecho de que se depositan diferentes cantidades al sembrar cada tipo de vegetal.

4. **Las reglas recursivas y conjuntas extra** que hemos tenido que implementar a través de la técnica de hechos ordenados, **no habrían sido necesarias si se hubiera utilizado otro lenguaje de programación**. Así, en lenguajes como Java, C o Python, una función recursiva sólo necesita la definición de una función y una acción conjunta se puede representar llamando a la segunda acción dentro de la función que represente a la primera.

5. **No se pueden definir estrategias sofisticadas con las herramientas que conocemos de CLIPS, al menos de manera sencilla**. De hecho, el haber incluido estrategias muy básicas en algunas de las fases del juego, sobre todo en `JORNADA-LABORAL`, ha incrementado muchísimo el número de reglas que hemos terminado implementando. Esto se debe principalmente al punto anterior, ya que muchas de las estrategias se basan en procesos recursivos.

6. **CLIPS sólo es capaz de resolver problemas relativamente sencillos**. Si no hubiéramos incluido más restricciones que las que se pedían en el enunciado (sobre todo la de quitar la ortogonalidad), habría sido muy complicado modelar este problema. De hecho, al intentar programar la acción de `construir-valla` con la estrategia que hemos explicado en el *Manual técnico* y además teniendo en cuenta la ortogonalidad, nos vimos incapaces de hacerlo sin usar menos de 30 o 40 reglas.

7. **Una de las grandes ventajas que ofrece CLIPS** frente a otros lenguajes de programación para este tipo de problemas e**s la representación del conocimiento**. La jerarquía de clases ha sido muy efectiva a la hora de modelar e incluso comprender *Agricola*. Lo más parecido que podemos encontrar en otros paradigmas es la programación orientada a objetos, que incluso puede llegar a implementar relaciones de jerarquía.

8. Por último, **concluir que los Sistemas de Producción podrían ser válidos para tareas que pueden resolverse con una sucesión de activaciones de reglas sencillas, que no involucren muchos operadores**, como es el caso de juegos de mesa como *Agricola*. Sin embargo, creemos que funciona mucho mejor cuanto más sencilla sea la estrategia de juego, como por ejemplo sucede en el Parchís o el Monopoly.

6. Comentarios personales

Esta práctica nos ha ayudado a entender la complejidad que hay detrás de los Sistemas de Producción, puesto que no hemos podido programar de igual manera que con otros lenguajes en los que ya estamos familiarizadas. Hemos encontrado dificultades, pero no en el entendimiento del uso de este nuevo lenguaje, sino en sus aplicaciones prácticas. Por otro lado, la abstracción de problemas que pueden parecer complejos en un principio porque no poseemos conocimiento sobre ellos (como en el caso de *Agricola*), hemos aprendido que sin embargo pueden resolverse mediante sucesiones sencillas de operadores.

Concluyendo, creemos que esta práctica se puede considerar un reto bastante instructivo en la materia si se intenta ser todo lo completo posible en cuanto a los diferentes detalles a tener en cuenta y si se busca que el sistema sea un poco inteligente siguiendo algún tipo de estrategia.